AD-A165 077 ADA (TRADEMARK) TRAINING CURRICULUM: ADVANCED ADA TOPICS L305 TEACHER'S GUIDE VOLUME 3(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K506 1/2

UNCLASSIFIED F/G 9/2 NL

1.0 4.5 5.0 5.6 2.8 3.2 3.6 4.0 2.5 2.2 2.0

1.1 1.8

1.25 1.4 1.6

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD-A165 077

1

1986

Ada® Training Curriculum

Advanced Ada® Topics
L305
Teacher's Guide
Volume III

Supersedes AD-A144 300

86 3 11 140

U.S. Army Communications-Electronics Command
(CECOM)

Contract DAAB07-83-C-K506

Prepared By:

SOFTECH, INC.
460 Totten Pond Road
Waltham, MA 02154

PART VI

LOW-LEVEL AND IMPLEMENTATION-DEPENDENT PROGRAMMING

SECTION 21

LOW-LEVEL AND IMPLEMENTATION-DEPENDENT FEATURES

INSTRUCTOR NOTES

SUCCEEDING SLIDES EXPAND ON THESE POINTS.

USE OF LOW-LEVEL FEATURES

- LOW-LEVEL FEATURES ALLOW ADA PROGRAMS TO INTERACT DIRECTLY WITH
  - HARDWARE
  - ASSEMBLY-LEVEL SOFTWARE REQUIRING SPECIFIC BIT-BY-BIT DATA REPRESENTATIONS
- LOW-LEVEL FEATURES SPECIFY DETAILS OF DATA REPRESENTATIONS AND ALGORITHMS, BUT STILL ALLOW THE DATA AND ALGORITHMS TO BE USED ABSTRACTLY
- LOW-LEVEL FEATURES SHOULD NOT BE USED ROUTINELY

INSTRUCTOR NOTES

BULLET 1: PROPERTIES INCLUDE MEMORY SIZE AND FLOATING POINT FORMAT, FOR EXAMPLE.

BULLET 3: THIS IS AN ESCAPE FROM THE STRONG TYPING RULES THAT USUALLY APPLY.

MOST OF THE TIME, WE DON'T WANT TO DO ANY OF THESE THINGS. OCCASIONALLY, WE HAVE TO IN ORDER TO INTERFACE WITH HARDWARE OR ASSEMBLY-LEVEL SOFTWARE DIRECTLY IN ADA.

SOME CAPABILITIES PROVIDED BY LOW-LEVEL FEATURES

- DETERMINE PROPERTIES OF THE MACHINE RUNNING THE PROGRAM.
- SPECIFY OR DETERMINE INTERNAL REPRESENTATION OF DATA.
- ADOPT THE ASSEMBLY-LANGUAGE VIEW OF THE SAME BITS REPRESENTING SEVERAL TYPES OF DATA AT ONCE.
- PERFORM RAW DEVICE-LEVEL I/O.

INSTRUCTOR NOTES

BULLET 2 REFERS TO THE FACT THAT THE PROGRAM'S LOGIC MAY BE DEPENDENT ON CERTAIN ASSUMPTIONS ABOUT THE HARDWARE OR THE OPERATING SYSTEM. BULLET 3 REFERS TO THE FACT THAT A PROGRAM USING LOW-LEVEL FEATURES MAY BE SYNTACTICALLY INCORRECT TO A DIFFERENT ADA COMPILER.

LOW-LEVEL FEATURES VARY FROM IMPLEMENTATION TO IMPLEMENTATION

- THIS REFLECTS VARIATIONS IN THE UNDERLYING TARGET COMPUTERS.
- USE OF LOW-LEVEL FEATURES GENERALLY MAKES PROGRAMS LESS PORTABLE, BY MAKING THEIR LOGIC DEPENDENT ON THE TARGET COMPUTER.
- RULES FOR CERTAIN LOW-LEVEL FEATURES WILL VARY FROM IMPLEMENTATION TO IMPLEMENTATION. SEE YOUR IMPLEMENTATION'S APPENDIX F.

INSTRUCTOR NOTES

AS LONG AS PROGRAMS OR PROGRAM COMPONENTS ARE CONCERNED WITH DEVICE- OR OPERATING-SYSTEM-LEVEL COMPUTATIONS, PORTABILITY IS NOT A REASONABLE GOAL ANYWAY.

INTERFACE MODULES ALLOW THE BULK OF THE SYSTEM, NOT CONCERNED WITH HARDWARE, TO REMAIN PORTABLE. TO MOVE THE PROGRAM TO ANOTHER MACHINE, ONLY THE INTERFACE MODULES NEED BE REPROGRAMMED.

INTERFACE MODULES AND INTERNAL VERSUS EXTERNAL STRUCTURE ARE ADDRESSED FURTHER IN THE NEXT TWO SLIDES.

GUIDELINES FOR USING LOW-LEVEL FEATURES

- USE THEM ONLY WHEN YOU MUST INTERACT WITH HARDWARE OR ASSEMBLY-LEVEL SOFTWARE.
- CONFINE THEIR USE TO A FEW PLACES IN THE PROGRAM, CALLED _INTERFACE MODULES_. INTERFACE MODULES PROVIDE THE REST OF THE PROGRAM WITH WAYS TO USE HARDWARE CAPABILITIES AT AN ABSTRACT LEVEL.
- AFTER DEFINING THE INTERNAL STRUCTURE OF DATA OR ALGORITHMS AT THE MACHINE LEVEL, USE THE DATA AND ALGORITHMS ACCORDING TO THEIR ABSTRACT EXTERNAL STRUCTURE.

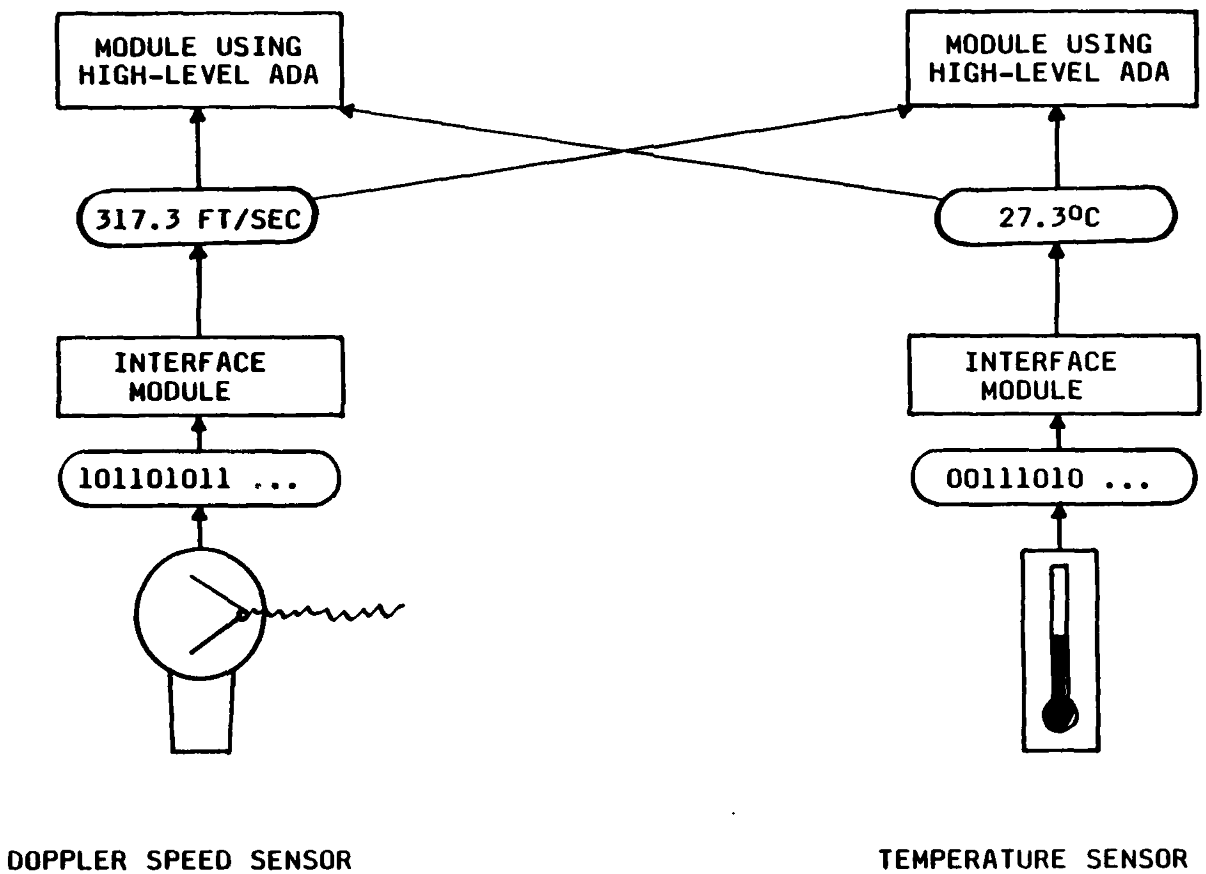
INSTRUCTOR NOTES

DEVICES WORK WITH SPECIFIC BIT PATTERNS THAT CAN ONLY BE SPECIFIED USING LOW-LEVEL FEATURES.

INTERFACE MODULES USE LOW-LEVEL FEATURES TO ALLOW DEVICES TO BE USED BY THE REST OF THE PROGRAM IN TERMS OF HIGH-LEVEL FEATURES.

THE DEPICTED MODULES USING HIGH-LEVEL ADA MAY THEMSELVES BE CONCERNED WITH DETAILED MANIPULATION OF THE DEVICES, BUT THE MANIPULATIONS CAN BE PERFORMED USING THE FEATURES OF ADA THAT THE CLASS HAS ALREADY LEARNED.

INTERFACE MODULES



VG 679.2

INSTRUCTOR NOTES

ADA PROVIDES THE MEANS TO SPECIFY BIT-BY-BIT REPRESENTATION ONCE AND THEN FORGET ABOUT IT WHEN USING THE DATA.

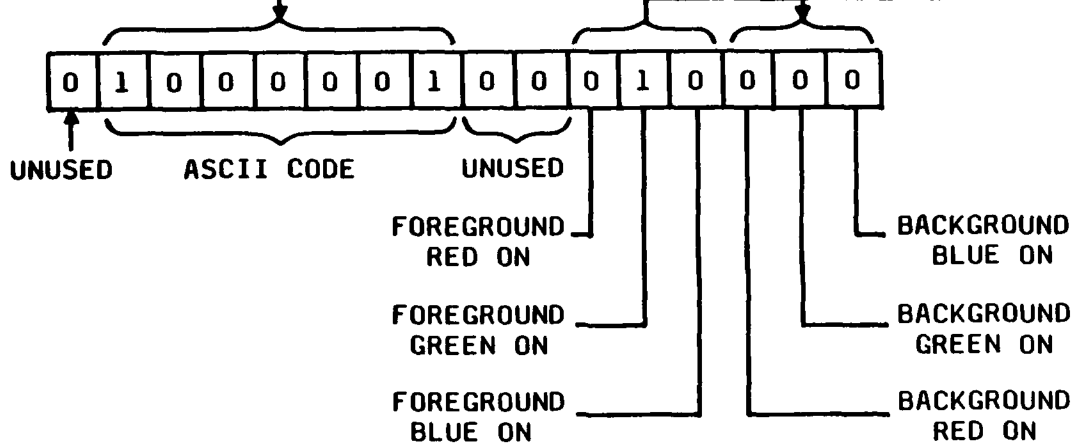
THE BOTTOM OF THE FIGURE DEPICTS A BIT-BY-BIT SPECIFICATION OF A 16-BIT WORD CONTROLLING THE CHARACTER DISPLAY, FOREGROUND COLOR, AND BACKGROUND COLOR OF ONE POSITION IN A COLOR DISPLAY. THE TYPE DECLARATIONS ABOVE PROVIDE AN ABSTRACT VIEW OF THE SAME INFORMATION. ONCE THE BIT-MAPPING OF RECORD COMPONENTS HAS BEEN SPECIFIED, PROGRAMMERS CAN USE Character_Display_Type AS THEY WOULD ANY RECORD TYPE, IRRESPECTIVE OF THE INTERNAL REPRESENTATION. AS SHOWN, NOT EVEN THE DECLARED ORDERING OF THE RECORD COMPONENTS NEED CORRESPOND TO THE INTERNAL ARRANGEMENT OF THE COMPONENTS.

INTERNAL VERSUS EXTERNAL STRUCTURE

```
type Primary_Color_Type is (Red, Green, Blue);

type Composite_Color_Type is
   array (Primary_Color_Type) of Boolean;

type Character_Display_Type is
   record
      Foreground_Color_Part : Composite_Color_Type;
      Background_Color_Part : Composite_Color_Type;
      Character_Part        : Character;
   end record;
```



INSTRUCTOR NOTES

AFTER ALL THESE FEATURES ARE PRESENTED, AN EXTENDED EXAMPLE OF Low_Level PROGRAMMING WILL BE GIVEN.

LOW-LEVEL FEATURES TO BE COVERED

- THE PACKAGE SYSTEM
- REPRESENTATION ATTRIBUTES
- PRAGMAS
- UNCHECKED CONVERSION
- INTERFACE WITH OTHER LANGUAGES
- DEVICE-LEVEL I/O PACKAGE

INSTRUCTOR NOTES

THE ITEMS PROVIDED BY ALL IMPLEMENTATIONS AND THE USE OF PRAGMAS TO ALTER THE DEFINITION OF SYSTEM ARE DISCUSSED ON SUBSEQUENT SLIDES.

THE PACKAGE SYSTEM

- A PREDEFINED PACKAGE PROVIDED BY EACH IMPLEMENTATION.
  - TO USE THE PACKAGE YOU MUST PROVIDE A with CLAUSE.
- DECLARES IMPLEMENTATION-DEFINED TYPES, SUBTYPES, AND NAMED NUMBERS.
- THE FOLLOWING ARE PROVIDED BY ALL IMPLEMENTATIONS OF SYSTEM:
  - NAMED NUMBERS INDICATING THE RANGE OF NUMERIC TYPE DECLARATIONS SUPPORTED BY THE IMPLEMENTATION
  - A TYPE FOR MACHINE ADDRESSES
  - AN ENUMERATION TYPE WHOSE VALUES CORRESPOND TO THE VARIOUS RUNTIME CONFIGURATIONS HANDLED BY THE IMPLEMENTATION
  - A CONSTANT OF THAT TYPE CORRESPONDING TO THE CURRENT TARGET MACHINE.
  - NAMED NUMBERS GIVING MACHINE-LEVEL CHARACTERISTICS OF THE SYSTEM (BITS PER STORAGE UNIT, MEMORY SIZE, CLOCK CYCLE TIME)
- AN IMPLEMENTATION MAY PROVIDE ADDITIONAL DECLARATIONS IN ITS OWN VERSION OF SYSTEM.
- CERTAIN CONSTANTS AND NAMED NUMBERS IN SYSTEM MAY BE CHANGED BY PRAGMAS.

INSTRUCTOR NOTES

A COMPILER THAT RUNS ON ONE MACHINE BUT GENERATES CODE FOR ANOTHER IS CALLED A <u>CROSS-COMPILER</u>. PROGRAMS FOR MICROCOMPUTERS ARE OFTEN COMPILED BY CROSS-COMPILERS RUNNING ON MAINFRAMES.

ENUMERATION TYPES WITH ONE VALUE ARE POSSIBLE. System.Name MAY BE SUCH A TYPE FOR SOME IMPLEMENTATIONS.

THE ADA REFERENCE MANUAL SAYS NOTHING WHATSOEVER ABOUT THE EFFECT OF SETTING System.System_Name TO A PARTICULAR VALUE. THAT IS ENTIRELY UP TO THE IMPLEMENTATION. THAT IS WHY THE SLIDE USES TERMS LIKE "SHOULD" AND "MAY".

ALTERNATIVE RUNTIME CONFIGURATIONS

- AN APSE MAY SUPPORT COMPILERS FOR SEVERAL TARGET MACHINES. THESE MAY BE VIEWED AS A SINGLE COMPILER THAT CAN BE DIRECTED TO PRODUCE OUTPUT FOR SEVERAL MACHINES.
- THE ENUMERATION TYPE System.Name CONTAINS ONE VALUE FOR EACH POSSIBLE TARGET MACHINE.
- THE CONSTANT System.System_Name IS A CONSTANT OF TYPE System.Name. IT SHOULD INDICATE THE TARGET MACHINE FOR WHICH THE COMPILER IS GENERATING MACHINE CODE.
- A SINGLE SOURCE PROGRAM REFERRING TO System.System_Name CAN BE COMPILED FOR SEVERAL TARGET MACHINES AND BE DESIGNED TO DO CERTAIN THINGS DIFFERENTLY FOR EACH TARGET.
- THE VALUE OF System.System_Name CAN BE SPECIFIED BY THE PRAGMA System_Name.
    - THE VALUE OF System.System_Name MAY DETERMINE THE WAY MACHINE CODE IS GENERATED.
    - THEN THE System_Name PRAGMA IS ESSENTIALLY A DIRECTIVE TO COMPILE CODE FOR A PARTICULAR TARGET MACHINE.

INSTRUCTOR NOTES

MACHINE ADDRESSES

- CERTAIN LOW-LEVEL FEATURES REFER TO SPECIFIC MACHINE ADDRESSES.

- THESE ADDRESSES ARE VALUES OF THE IMPLEMENTATION-DEFINED TYPE System.Address.

- A FEW OF THE POSSIBLE DEFINITIONS OF System.Address (DEPENDING ON THE TARGET ARCHITECTURE):
  - A NON-NEGATIVE INTEGER
  - A "SEGMENT NAME" PLUS AN OFFSET WITHIN A SEGMENT
  - (FOR A DISTRIBUTED SYSTEM) A "SITE NAME" PLUS AN ADDRESS INTO A PARTICULAR SITE'S STORAGE

- THE NAMED NUMBER System.Storage_Unit GIVES THE SIZE IN BITS OF THE STORAGE UNIT NAMED BY AN ADDRESS.
  - IF EACH 8-BIT BYTE HAS ITS OWN ADDRESS, System.Storage_Unit = 8
  - IF EACH ADDRESS CORRESPONDS TO A DIFFERENT 36-BIT WORD, System.Storage_Unit = 36

- THE NAMED NUMBER System.Memory_Size GIVES THE NUMBER OF STORAGE UNITS IN THE RUNTIME CONFIGURATION.

- System.Storage_Unit AND System.Memory_Size CAN BE CHANGED BY PRAGMAS.

INSTRUCTOR NOTES

THE LIMITS OF FIXED-POINT TYPE DECLARATIONS ARE DEFINED BY A TRADEOFF BETWEEN PRECISION AND RANGE.

THE COMPILE-TIME CHECK EXPLAINED IN THE LAST BULLET IS A GOOD REASON FOR DEFINING NEW NUMERIC TYPES BASED ON EXPECTED REQUIREMENTS INSTEAD OF USING PREDEFINED TYPES. WHEN MOVING TO A NEW TARGET MACHINE, YOU WILL BE WARNED AT COMPILE TIME OF ANY NUMERIC RANGE OR PRECISION PROBLEMS.

NUMERIC CAPABILITIES OF AN IMPLEMENTATION

- System.Min_Int AND System.Max_Int GIVE THE LOWER AND UPPER BOUNDS OF THE INTEGER TYPES SUPPORTED BY AN IMPLEMENTATION. THE RANGE SPECIFIED IN ANY INTEGER TYPE DECLARATION MUST BE CONTAINED WITHIN THE RANGE.

  ```
  System.Min_Int .. System.Max_Int
  ```

- System.Max_Digits SPECIFIES THE MAXIMUM NUMBER OF SIGNIFICANT DIGITS THAT CAN BE SPECIFIED IN A FLOATING POINT TYPE DECLARATION.

- System.Max_Mantissa SPECIFIES THE MAXIMUM VALUE OF a+b ALLOWED FOR A FIXED-POINT TYPE DECLARATION OF THE FORM

  type __________ IS DELTA 2.0^{-a} RANGE $-(2.0^{b}-1)$ .. $(2.0^{b}-1)$;

  (WHEN THE DELTA AND RANGE ARE NOT OF THIS FORM, THE NEXT-HIGHER VALUES OF a OR b ARE USED.)

- System.Fine_Delta SPECIFIES THE MINIMUM DELTA THAT CAN BE SPECIFIED IN A FIXED-POINT TYPE DECLARATION WITH THE RANGE -1.0 .. 1.0 (ALWAYS EQUAL TO 2.0**(-System.Max_Mantissa)).

- THERE ARE NO LANGUAGE-DEFINED PRAGMAS TO CHANGE THESE NAMED NUMBERS DIRECTLY, BUT FOR CERTAIN IMPLEMENTATIONS THEY MAY BE CHANGED INDIRECTLY WHEN System.System_Name IS REDEFINED.

- A TYPE DECLARATION EXCEEDING AN IMPLEMENTATION'S CAPABILITIES CAUSES A COMPILE-TIME ERROR.

INSTRUCTOR NOTES

AVOID PROLONGED DISCUSSION OF System.Tick, WHICH IS OF INTEREST PRIMARILY TO REAL-TIME PROGRAMMERS.

THE RELATIONSHIP BETWEEN CYCLE TIME AND EXECUTION SPEED ALSO DEPENDS ON THE AMOUNT OF WORK ACCOMPLISHED DURING A CYCLE. THIS IS A FUNCTION OF THE INSTRUCTION SET.

PROCESSOR CLOCK CYCLE

- THE NAMED NUMBER System.Tick GIVES THE TARGET PROCESSOR'S BASIC CLOCK CYCLE IN SECONDS.
- A PROGRAM REFERRING TO System.Tick CAN BE MADE DEPENDENT ON THE SPEED OF THE TARGET PROCESSOR.
  - GENERALLY SPEAKING, SHORTER CYCLE = FASTER EXECUTION
- System.Tick CANNOT BE CHANGED DIRECTLY BY A LANGUAGE-DEFINED PRAGMA.

INSTRUCTOR NOTES

A COMPILATION IS SIMPLY A SEQUENCE OF COMPILATION UNITS SUBMITTED TO THE COMPILER AT ONE TIME.

USE OF PRAGMAS TO ALTER THE DEFINITION OF SYSTEM

- THREE LANGUAGE-DEFINED PRAGMAS

```
pragma System_Name (enumeration literal of type System.Name);
          -- alters System.System_Name
     pragma Storage_Unit (integer literal);
          -- alters System.Storage_Unit
     pragma Memory_Size (integer literal);
          -- alters System.Memory_Size
```

- EFFECT IS AS IF THE PACKAGE SYSTEM WERE MODIFIED AND RECOMPILED.
  - PREVIOUSLY COMPILED COMPILATION UNITS WITH A with CLAUSE FOR SYSTEM MUST THEN BE RECOMPILED.
  - AN ALTERNATIVE VERSION OF THE PACKAGE RESULTS.

- RESTRICTIONS ON THE USE OF THE PRAGMAS.
  - ONLY ALLOWED BEFORE THE FIRST COMPILATION UNIT OF A COMPILATION.
  - AN IMPLEMENTATION MAY IMPOSE FURTHER RESTRICTIONS, SUCH AS ONLY ALLOWING THEM IN THE INITIAL COMPILATION AFTER A NEW PROGRAM LIBRARY IS CREATED.

INSTRUCTOR NOTES

IN THE TERMS GIVEN THREE SLIDES EARLIER, b=10 SINCE 2**10=1024 IS THE NEXT POWER OF TWO ABOVE 1000.0. THUS 10-System.Max_Mantissa = b-(a+b) = -1. THE MACHINE DEPENDENT VERSION ASSUMES THAT System.Max_Mantissa = 31. (2#1.0#E-21 = 1.0*2.0**(-21).)

ABSTRACT DESCRIPTION OF CONFIGURATION DEPENDENCIES

- THE PACKAGE SYSTEM ALLOWS CONFIGURATION-DEPENDENT VALUES TO BE NAMED ABSTRACTLY.
- PROGRAMMERS NEED NOT MEMORIZE ACTUAL VALUES.
- THE SAME SOURCE PROGRAM CAN BE COMPILED FOR DIFFERENT TARGETS, AND THE COMPILER WILL SUBSTITUTE THE PROPER VALUES IN EACH CASE.
- THE ABSTRACT FORMULATION OF CONFIGURATION DEPENDENCIES <u>INCREASES</u> PORTABILITY.
- TWO WAYS TO DECLARE A FIXED POINT TYPE WITH RANGE -1000.0 .. 1000.0 AND THE MAXIMUM PRECISION SUPPORTED BY THE IMPLEMENTATION:

  <u>A PROGRAMMER FAMILIAR WITH HIS IMPLEMENTATION MIGHT WRITE</u>:

  ```
  delta 2#1.0#E-21 range -1000.0 .. 1000.0
  ```

  <u>A PROGRAMMER INTERESTED IN READABILITY AND PORTABILITY SHOULD WRITE</u>:

  ```
  delta 2.0**(10-System.Max_Mantissa) range -1000.0 .. 1000.0
  ```

INSTRUCTOR NOTES

A PROGRAMMER WHO "KNOWS" THAT HIS IMPLEMENTATION STORES RECORD COMPONENTS IN ORDER OF COMPONENT DECLARATION MAY BE IN FOR A SHOCK WHEN THE NEXT RELEASE OF THE COMPILER COMES OUT.

BY DESCRIBING POSITIONS OF RECORD COMPONENTS IN TERMS OF REPRESENTATION ATTRIBUTES RATHER THAN NUMBERS, HE IS RELYING ON RULES THAT ARE GUARANTEED NOT TO CHANGE. HIS PROGRAM IS ALSO MORE READILY UNDERSTOOD.

REPRESENTATION ATTRIBUTES

- REPRESENTATION ATTRIBUTES ARE ATTRIBUTES RETURNING MACHINE-LEVEL CHARACTERISTICS OF PROGRAMS AND DATA.
- LOW LEVEL - ONLY APPROPRIATE IN SPECIAL CONTEXTS.
- ABSTRACT DESCRIPTION OF THE SIZE OF AN OBJECT, FOR EXAMPLE, CAN REDUCE DEPENDENCY ON A PARTICULAR IMPLEMENTATION, CONFIGURATION, OR INTERNAL COMPILER DESIGN DECISION.

INSTRUCTOR NOTES

X'Address MAY RETURN THE ADDRESS OF A DATA CELL OR THE ADDRESS OF A MACHINE-LANGUAGE INSTRUCTION.

IF X IS AN OBJECT OF TYPE T, T'Size ≤ X'Size. (AN INDIVIDUAL OBJECT MAY CONTAIN EXTRA BITS.)

THERE IS A COLLECTION OF STORAGE ASSOCIATED WITH EACH ACCESS TYPE FOR CREATION OF ALLOCATED VARIABLES. DIFFERENT ACCESS TYPE DECLARATIONS HAVE DIFFERENT COLLECTIONS

EMPHASIZE THAT 'Size IS IN BITS, 'Storage_Size IS IN STORAGE UNITS (BYTES, WORDS, OR WHATEVER AN ADDRESS REFERS TO).

ATTRIBUTES FOR SIZES AND ADDRESSES

- X'Address
  - X MAY BE A DATA OBJECT, A SUBPROGRAM, OR A STATEMENT LABEL, AMONG OTHER THINGS
  - THE VALUES OF X'Address IS OF TYPE System.Address
  - X'Address IS ALLOWED EVEN IF X IS AN OBJECT IN A PRIVATE TYPE
- X'Size
  - X MAY BE A DATA OBJECT, A TYPE, OR A SUBTYPE
  - RETURNS THE NUMBER OF BITS ALLOCATED TO HOLD A PARTICULAR OBJECT, OR *THE MINIMUM NUMBER USED BY* THE IMPLEMENTATION FOR OBJECTS IN A PARTICULAR TYPE OR SUBTYPE
  - IF X IS AN ACCESS VALUE, X'Size IS THE SIZE OF THE POINTER AND X.all'Size IS THE SIZE OF THE ALLOCATED OBJECT IT POINTS TO
  - X'Size IS ALLOWED EVEN IF X IS AN OBJECT IN A PRIVATE TYPE
- T'Storage_Size
  - T MAY BE AN ACCESS TYPE OR SUBTYPE OF AN ACCESS TYPE
  - RETURNS THE NUMBER OF STORAGE UNITS RESERVED FOR ALLOCATED VARIABLES TO BE DESIGNATED BY VALUES OF THE SPECIFIED ACCESS TYPE.

INSTRUCTOR NOTES

THESE ATTRIBUTES MAY VARY FROM OBJECT TO OBJECT IN A RECORD TYPE WITH DISCRIMINANTS.

EMPHASIZE THAT 'Position USES STORAGE UNITS WHILE 'First_Bit AND 'Last_Bit USE BITS.

ATTRIBUTES FOR RECORD LAYOUT

- LET R BE AN OBJECT IN A RECORD TYPE WITH COMPONENT C.
- R.C'Position IS THE OFFSET FROM THE FIRST STORAGE UNIT OCCUPIED BY R OF THE FIRST STORAGE UNIT OCCUPIED BY R.C. <u>MEASURED IN STORAGE UNITS</u>.
- R.C'First_Bit IS THE OFFSET FROM THE START OF THE STORAGE UNIT GIVEN BY R.C'Position OF THE FIRST BIT OCCUPIED BY R.C. <u>MEASURED IN BITS</u>.
- R.C'Last_Bit IS THE OFFSET FROM THE START OF THE STORAGE UNIT GIVEN BY R.C'Position OF THE LAST BIT OCCUPIED BY R.C. <u>MEASURED IN BITS</u>.

INSTRUCTOR NOTES

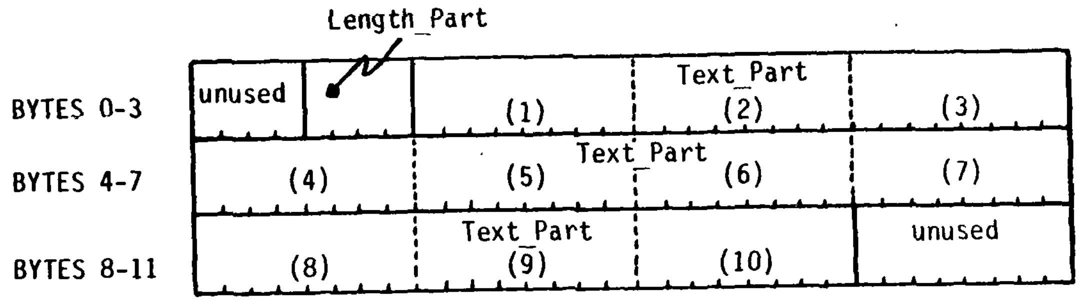
OBSERVE THAT 'Last_Bit MAY BE HIGHER THAN THE NUMBER OF BITS IN ONE STORAGE UNIT.

EXAMPLE OF RECORD LAYOUT ATTRIBUTES

- RECORD TYPE DECLARATION:

```
type Name_Type is
   record
      Length_Part : Integer range 0 .. 10;
      Text_Part   : String (1 .. 10);
   end record;
```

- ASSUME THE ADDRESSABLE STORAGE UNIT IS AN 8-BIT BYTE AND THE COMPILER MAPS STORAGE AS FOLLOWS:



INSTRUCTOR NOTES

ATTRIBUTE VALUES

ATTRIBUTE VALUES (ASSUMING NAME IS OF TYPE Name_Type)

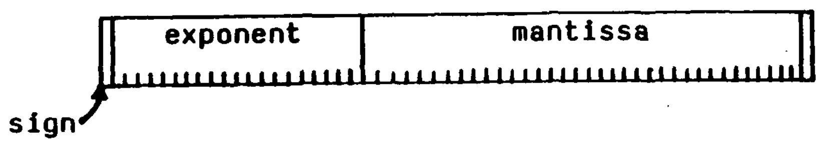
```
Name.Length_Part'Position  = 0      (byte 0)
Name.Length_Part'First_Bit = 4      (bit 4)
Name.Length_Part'Last_Bit  = 7      (bit 7)

Name.Text_Part'Position    = 1      (byte 1)
Name.Text_Part'First_Bit   = 0      (bit 0)
Name.Text_Part'Last_Bit    = 79     (bit 79 from start of byte)
```

INSTRUCTOR NOTES

AVOID A PROLONGED DISCUSSION OF MACHINE FLOATING-POINT REPRESENTATIONS. THESE FEATURES ARE OF USE ONLY TO THOSE WHO ALREADY UNDERSTAND SUCH MATTERS.

A TYPICAL FLOATING POINT FORMAT IS AS FOLLOWS:



A 7-BIT EXPONENT TYPICALLY HOLDS VALUES FROM -64 TO 63, EITHER IN TWOS-COMPLEMENT NOTATION OR BY TREATING THE 7 BITS AS AN UNSIGNED INTEGER AND SUBTRACTING 64.

THE MANTISSA IS VIEWED EITHER AS A SET OF N BINARY DIGITS OR N/4 HEXADECIMAL DIGITS PRECEDED BY A RADIX POINT. (IN THIS EXAMPLE N=24).

THE MAGNITUDE OF THE FLOATING-POINT NUMBER IS OBTAINED BY RAISING THE RADIX (EITHER 2 or 16) TO THE POWER OF THE EXPONENT AND THE MULTIPLYING BY THE MANTISSA.

CERTAIN OPERATIONS MAY REQUIRE THAT THE MANTISSA BE NORMALIZED, I.E. THAT IT BE SHIFTED LEFT, AND THE EXPONENT DECREASED, UNTIL THE FIRST NON-ZERO DIGIT (IF ANY) IS IN THE LEFT MOST POSITION. FOR RADIX 16, THIS MEANS THAT THE FIRST HEXADECIMAL DIGIT IS NON-ZERO, BUT THE FIRST ONE, TWO, OR THREE BITS OF THE FIRST HEXADECIMAL DIGIT MAY BE ZERO.

REPRESENTATION ATTRIBUTES OF REAL TYPES

- ASSUMING T IS A FIXED- OR FLOATING POINT TYPE OR SUBTYPE, THE FOLLOWING ATTRIBUTES HAVE VALUES OF TYPE BOOLEAN:

  - T'Machine_Rounds: INDICATES WHETHER EVERY ARITHMETIC OPERATION THAT CANNOT DELIVER AN EXACT RESULT ROUNDS RATHER THAN TRUNCATES.

  - T'Machine_Overflows: TRUE IF THE HARDWARE DETECTS OVERFLOW AND RAISES *Numeric_Error* FOR OPERATIONS OF TYPE T THAT OVERFLOW. (AN IMPLEMENTATION IS NOT REQUIRED TO DETECT OVERFLOW.)

- ASSUMING T IS A FLOATING-POINT TYPE OR SUBTYPE, THE FOLLOWING ATTRIBUTES CAN BE USED AS IF THEY WERE INTEGER LITERALS:

  - T'Machine_Radix: THE RADIX USED IN THE MACHINE REPRESENTATION FOR TYPE T. (TYPICALLY 2 OR 16 AFFECTS THE MEANING OF THE INTERNAL EXPONENT AND THE DEFINITION OF A "NORMALIZED" FLOATING POINT NUMBER.)

  - T'Machine_Mantissa: THE NUMBER OF DIGITS IN BASE T'Machine_Radix IN THE MANTISSA OF THE MACHINE REPRESENTATION FOR TYPE T.

  - T'Machine_EMax: THE LARGEST VALUE FOR THE EXPONENT OF THE MACHINE REPRESENTATION FOR TYPE T.

  - T'Machine_EMin: THE MOST NEGATIVE VALUE FOR THE EXPONENT OF THE MACHINE REPRESENTATION FOR TYPE T.

INSTRUCTOR NOTES

THE NEXT SLIDE GIVES THE FORM FOR EACH KIND OF REPRESENTATION CLAUSE.

THE FIFTH BULLET BEGINS "USUALLY" BECAUSE REPRESENTATION CLAUSES CAN ALSO GO IN TASK SPECIFICATIONS.

CERTAIN OCCURRENCES OF A TYPE NAME, E.G. IN AN OBJECT DECLARATION, REQUIRE KNOWLEDGE OF *THE TYPES REPRESENTATION. OTHER* OCCURRENCES, E.G. IN DECLARATIONS OF OTHER TYPES, DO NOT. THE FULL LIST IS GIVEN IN SECTION 13.1, PARAGRAPH 6 OF THE REFERENCE MANUAL, WHERE THE OCCURRENCES REQUIRING KNOWLEDGE OF THE REPRESENTATION ARE CALLED "FORCING OCCURRENCES."

REPRESENTATION CLAUSES

- SPECIFY INTERNAL REPRESENTATIONS
- MAY DETERMINE THE VALUES OF REPRESENTATION ATTRIBUTES
- LOW-LEVEL -- ONLY APPROPRIATE IN CERTAIN CONTEXTS
- GENERAL FORM:

  FOR ________________ USE ______________;

- USUALLY GOES IN THE SEQUENCE OF DECLARATIONS IN A DECLARATIVE PART OR IN A PACKAGE DECLARATION.
  - MUST BE IN THE SAME PROGRAM UNIT AS THE DECLARATION OF THE ENTITY WHOSE REPRESENTATION IT SPECIFIES, PRIOR TO ANY USE OF THE ENTITY THAT REQUIRES KNOWLEDGE OF ITS REPRESENTATION.
  - IN A PACKAGE SPECIFICATION, IT IS GOOD PRACTICE TO PLACE REPRESENTATION CLAUSES IN THE PRIVATE PART (EVEN IF THERE ARE NO PRIVATE TYPES) SINCE THEY DESCRIBE THE IMPLEMENTATION RATHER THAN THE EXTERNAL INTERFACE OF TYPES DECLARED IN THE VISIBLE PART.
  - IN A DECLARATIVE PART, REPRESENTATION SPECIFICATIONS MUST PRECEDE SUBPROGRAM, PACKAGE, AND TASK BODIES OR BODY STUBS.
- MAY BE REJECTED BY THE COMPILER, MAKING THE PROGRAM ILLEGAL.

INSTRUCTOR NOTES

A LENGTH CLAUSE CONTROLS THE SIZE OF THE OBJECTS IN A TYPE, THE AMOUNT OF STORAGE TO BE MADE AVAILABLE FOR ALLOCATED VARIABLES, OR THE DISTANCE BETWEEN EXACTLY REPRESENTED VALUES IN A FIXED-POINT TYPE.

AN ENUMERATION REPRESENTATION CLAUSE SPECIFIES THE INTERNAL INTEGER ENCODINGS FOR VALUES IN AN ENUMERATION TYPE.

A RECORD REPRESENTATION CLAUSE SPECIFIES THE BIT-BY-BIT POSITIONING OF INDIVIDUAL COMPONENTS WITHIN A RECORD, AS WELL AS THE STORAGE BOUNDARY ALIGNMENT OF THE RECORD AS A WHOLE.

AN ADDRESS CLAUSE CAN SPECIFY THE ADDRESS OF AN ENTITY (OR ASSOCIATE A HARDWARE INTERRUPT WITH A TASK ENTRY).

A LATER SLIDE GIVES THE FORM OF A RECORD REPRESENTATION CLAUSE.

FORMS OF REPRESENTATION CLAUSES

TYPE REPRESENTATION CLAUSES

LENGTH CLAUSE:

for $\boxed{\text{type name}}$' $\left\{\begin{array}{l}\text{Size}\\ \text{Storage\_Size}\\ \text{Small}\end{array}\right\}$ use $\boxed{\text{expression}}$;

ENUMERATION REPRESENTATION CLAUSE:

for $\boxed{\text{enumeration type name}}$ use $\boxed{\text{one-dimensional array aggregate}}$;

RECORD REPRESENTATION CLAUSE:

for $\boxed{\text{record type name}}$ use $\boxed{\text{record representation clause}}$;

ADDRESS CLAUSES

for $\boxed{\text{identifier}}$ use at $\boxed{\text{expression of type System.Address}}$;

INSTRUCTOR NOTES

THE NUMBER OF BITS NEED NOT BE A STATIC EXPRESSION.

AN IMPLEMENTATION NEED NOT, AND PROBABLY WON'T, USE A LENGTH CLAUSE FOR TYPE T IN SELECTING A REPRESENTATION FOR COMPONENTS OF TYPE T. RATHER, A LENGTH CLAUSE IS MEANT TO AFFECT HOW THE COMPONENTS WILL BE FIT TOGETHER, GIVEN THAT THEIR REPRESENTATION HAS ALREADY BEEN SELECTED.

SPECIFYING THE MAXIMUM SIZE OF OBJECTS IN A TYPE

- LENGTH CLAUSE: for [type name]'Size use [number of bits];

- SPECIFIES AN UPPER BOUND FOR THE NUMBER OF BITS TO BE USED FOR OBJECTS IN THE NAMED TYPE.

  (AS AN ATTRIBUTE, T'Size RETURNS THE LOWER BOUND ON THE NUMBER OF BITS REQUIRED FOR OBJECTS OF TYPE T.)

- EVERY POSSIBLE VALUE OF THE TYPE MUST FIT IN THE SPECIFIED NUMBER OF BITS.

- MAY AFFECT THE PRESENCE, ABSENCE, OR NUMBER OF UNUSED BITS BETWEEN COMPONENTS OF AN ARRAY OR RECORD TYPE.

INSTRUCTOR NOTES

EMPHASIZE THAT 'Size IS IN TERMS OF BITS, 'Storage_Size IN TERMS OF STORAGE UNITS.

INTERNAL CONTROL INFORMATION MENTIONED IN THE THIRD BULLET MAY BE USED TO KEEP TRACK OF WHICH STORAGE IS ALREADY ALLOCATED AND WHICH IS AVAILABLE FOR ALLOCATION.

(THIS CLAUSE MAY ALSO APPEAR WITH A TASK TYPE NAME IN PLACE OF THE ACCESS TYPE NAME. IT THEN SPECIFIES THE NUMBER OF STORAGE UNITS TO BE RESERVED FOR TASK ACTIVATIONS.)

SPECIFYING THE AMOUNT OF STORAGE AVAILABLE FOR ALLOCATED OBJECTS

- LENGTH CLAUSE: for access type name 'Storage_Size use number of storage units ;

- SPECIFIES THE NUMBER OF STORAGE UNITS TO BE RESERVED FOR ALLOCATED VARIABLES DESIGNATED BY THE ACCESS TYPE.

- DEPENDING ON THE IMPLEMENTATION, THIS MAY INCLUDE STORAGE USED FOR INTERNAL CONTROL INFORMATION ASSOCIATED WITH ALLOCATED OBJECTS.

- ASSUMING THAT STORAGE FOR CONTROL INFORMATION IS NOT TAKEN FROM THAT SPECIFIED BY THE LENGTH CLAUSE, THAT ONLY WHOLE NUMBERS OF STORAGE UNITS ARE ALLOCATED AT ONCE AND THAT ALL OBJECTS IN Designated_Type ARE THE SAME SIZE, THE FOLLOWING DECLARATIONS WILL ALLOW Number_of_Allocations VARIABLES OF TYPE Designated_Type TO BE DYNAMICALLY ALLOCATED:

```
Bits_Per_Unit    : constant := System.Storage_Unit;
Bits_Per_Object  : constant := Designated_Type'Size;
Units_Per_Object : constant := (Bits_Per_Object + Bits_Per_Unit-1)/Bits_Per_Unit;
   -- The integer quotient (a+b-1)/b is the ceiling of the real number a/b.
type Access_Type is access Designated_Type;
for Access_Type'Storage_Size use Number_of_Allocations * Units_Per_Object;
```

INSTRUCTOR NOTES

THERE MUST BE EXACT REPRESENTATIONS OF EACH NUMBER OF THE FORM n*T'Small FOR n IN THE APPROPRIATE RANGE. THE ALLOWABLE OPERATIONS ON FIXED POINT TYPES ARE SUCH THAT THE MOST REASONABLE INTERNAL REPRESENTATION FOR n*T'Small IS THE NUMBER n ITSELF. INTERNAL DELTAS THAT ARE EXACT POWERS OF A FACILITATE EFFICIENT CONVERSION FROM ONE FIXED-POINT TYPE TO ANOTHER, BUT MAY NOT BE WHAT THE PROGRAMMER REALLY WANTS.

SPECIFYING EXACTLY REPRESENTED VALUES OF A FIXED-POINT TYPE

- LENGTH CLAUSE: for `fixed-point type name`'Small use `increment`;

- THE INCREMENT MAY BELONG TO ANY REAL TYPE, BUT MUST NOT BE GREATER THAN THE DELTA SPECIFIED IN THE FIXED POINT TYPE DECLARATION.

- AN IMPLEMENTATION MUST PROVIDE EXACT REPRESENTATIONS FOR INTEGER MULTIPLES OF SOME "INTERNAL DELTA."

  - THE DEFAULT VALUE OF THE INTERNAL DELTA IS A VALUE OF THE FORM 1.0/2.0 N LESS THAN OR EQUAL TO THE "EXTERNAL DELTA" SPECIFIED IN THE FIXED-POINT TYPE DECLARATION.

  - THE LENGTH CLAUSE ALLOWS SPECIFICATION OF A NEW INTERNAL DELTA.

  - THE INTERNAL DELTA OF A FIXED-POINT TYPE T IS GIVEN BY THE ATTRIBUTE T'Small.

- EXAMPLE:

  ```
  type Money_Type is delta 0.01 range 0.1 .. 1.0E4;
  ```

  BY DEFAULT, THIS MEASURES MONEY NOT IN CENTS, BUT IN 128ths OF A DOLLAR.

  THE FOLLOWING REPRESENTATION CLAUSE FIXES THIS:

  ```
  for Money_Type'Small use 0.01;
  ```

INSTRUCTOR NOTES

THE FACT THAT THE ARRAY IS IN ASCENDING ORDER MEANS THAT THE ORDERING OF INTEGER ENCODINGS IS CONSISTENT WITH THE ABSTRACT ORDERING OF ENUMERATION TYPE VALUES.

AS SHOWN, THE ENCODING MAY CONTAIN GAPS. HOWEVER, THIS MAKES CERTAIN NORMALLY QUITE EFFICIENT OPERATIONS LESS EFFICIENT -- 'Succ, 'Pred, 'Pos (WHICH CONTINUES TO EVALUATE TO THE ABSTRACT POSITION NUMBER RATHER THAN THE INTERNAL CODING), AND ARRAY INDEXING.

THOUGH A NAMED AGGREGATE IS USED ON THE SLIDE, POSITIONAL AGGREGATES ARE ALSO LEGAL. A NAMED AGGREGATE IS PREFERABLE BECAUSE IT CLEARLY IDENTIFIES THE ENCODING OF EACH ENUMERATION LITERAL.

SPECIFYING ENCODINGS OF ENUMERATION TYPE VALUES

- ENUMERATION REPRESENTATION CLAUSE:

  for $\boxed{\text{enumeration type name}}$ use $\boxed{\text{one-dimensional array aggregate}}$;

- ARRAY IS INDEXED BY THE ENUMERATION TYPE. COMPONENTS ARE SPECIFIED BY STATIC UNIVERSAL INTEGER EXPRESSING (TYPICALLY INTEGER LITERALS).

- THE COMPONENT INDEXED BY ENUMERATION VALUE x IS THE INTEGER ENCODING USED AS THE INTERNAL REPRESENTATION OF x.

- THE ARRAY MUST BE IN STRICT ASCENDING ORDER (NO DUPLICATION).

- EXAMPLE: A CARD READER PROVIDES THE FOLLOWING STATUS CODES:

0	Character ready for transmission
1	Mechanical error
2	Illegal punch
3	unused
4	Waiting for data
5	unused
6	unused
7	unused
8	Buffer overwritten before transmission

```
type Card_Reader_Status_Type is
   (Character_Ready, Mechanical_Error, Illegal_Punch, Waiting, Data_Lost);
for Card_Reader_Status_Type use
   (Character_Ready => 0, Mechanical_Error => 1, Illegal_Punch => 2,
   Waiting => 4, Data_Lost => 8);
```

NOW THE REST OF THE PROGRAM CAN BE WRITTEN AS THOUGH THE CARD READER DIRECTLY PROVIDES THESE FIVE ENUMERATION VALUES, WITHOUT REFERENCE TO THE NUMERIC CODES.

INSTRUCTOR NOTES

AN EXAMPLE FOLLOWS.

SPECIFYING BIT-BY-BIT ARRANGEMENT OF RECORD COMPONENTS

- <u>RECORD REPRESENTATION CLAUSE</u>:

```
for [record type name] use
   record [at mod [storage unit multiple];]
      {[component name] at [storage unit] range [bit range];}
   end record;
```

- STORAGE BOUNDARY ALIGNMENT:

  - [storage unit multiple] IS A STATIC EXPRESSION OF ANY INTEGER TYPE
  - ALL RECORDS IN THE TYPE MUST START AT AN ADDRESS THAT IS AN EXACT MULTIPLE OF THE GIVEN VALUE
  - FOR A BYTE-ADDRESSABLE MACHINE WITH 4-BYTE WORDS (E.G. THE IBM 370):

    at mod 2 SPECIFIES HALF-WORD ALIGNMENT
    at mod 4 SPECIFIES FULL-WORD ALIGNMENT
    at mod 8 SPECIFIES DOUBLE-WORD ALIGNMENT

INSTRUCTOR NOTES

SPECIFYING BIT-BY-BIT ARRANGEMENT OF RECORD COMPONENTS (CONTINUED)

- COMPONENT OFFSET FROM START OF RECORD
  - $\boxed{\text{storage unit}}$ IS A STATIC EXPRESSION OF ANY INTEGER TYPE.
  - $\boxed{\text{bit range}}$ IS A STATIC RANGE (E.G. 0 .. 3) THAT MAY HAVE TWO BOUNDS BELONGING TO DIFFERENT INTEGER TYPES.
  - THE FIRST STORAGE UNIT OF THE RECORD IS NUMBERED ZERO.
  - PHYSICAL ORDERING OF COMPONENTS NEED NOT MATCH THE ORDER IN THE RECORD TYPE DECLARATION.
  - BOTH DISCRIMINANTS AND ORDINARY COMPONENTS MAY BE SPECIFIED.
  - SOME COMPONENTS MAY BE LEFT UNSPECIFIED.
  - OVERLAPPING COMPONENTS ALLOWED IF AND ONLY IF THEY ARE IN DIFFERENT VARIANTS.
- MANY IMPLEMENTATION VARIATIONS -- SEE YOUR COMPILER'S REFERENCE MANUAL (APPENDIX F)
  - ALLOWABLE ALIGNMENTS (IF ANY)
  - ARE BITS NUMBERED LEFT-TO-RIGHT OR RIGHT-TO-LEFT?
  - MAY BIT NUMBERING EXTEND TO ADJACENT STORAGE UNITS?
  - MAY COMPONENTS CROSS STORAGE UNIT BOUNDARIES?

INSTRUCTOR NOTES

IN THE Data_Address AND Byte_Count FIELDS, HEXADECIMAL LITERALS PROVIDE THE CLEAREST WAY OF SAYING "THE LARGEST UNSIGNED INTEGER THAT CAN FIT IN THREE BYTES AND ... IN TWO BYTES." UNDERSCORES ARE USED TO GROUP HEXADECIMAL DIGITS INTO BYTES.

EXAMPLE OF RECORD REPRESENTATION

- IBM 370 CHANNEL COMMAND WORD (REALLY A DOUBLE-WORD):

COMMAND CODE	DATA ADDRESS		
0 7	8 15	16 23	24 31

FLAGS	UNUSED	BYTE COUNT	
32 39	40 47	48 55	56 63

BIT 32: CHAIN DATA
BIT 33: CHAIN COMMAND
BIT 34: SUPPRESS LENGTH INDICATION
BIT 35: SKIP
BIT 36: PROGRAM CONTROLLED INTERRUPT
BIT 37: INDIRECT ADDRESS
BIT 38: UNUSED
BIT 39: UNUSED

INSTRUCTOR NOTES

EXAMPLE OF RECORD REPRESENTATION (CONTINUED)

- ABSTRACT RECORD TYPE DECLARATION:

```
type CCW_Type is
   record
      Command_Code : Command_Code_Type;
      Data_Address : System.Address range 0 .. 16#FF_FF_FF#;
      Byte_Count   : Integer range 0 .. 16#FF_FF#;
      Chain_Data_Flag,
         Chain_Command_Flag,
         Suppress_Length_Indication_Flag,
         Skip_Flag,
         Program_Controlled_Interrupt_Flag,
         Indirect_Address_Flag : Boolean := False;
   end record;
```

INSTRUCTOR NOTES

THE "AT MOD 8" FORCES DOUBLE-WORD ALIGNMENT FOR EACH CCW_Type OBJECT.

RANGES LIKE 0 .. 0, 1 .. 1, AND 2 .. 2 SPECIFY THAT A COMPONENT IS TO OCCUPY A SINGLE BIT. (THIS RECORD REPRESENTATION CLAUSE CAN ONLY BE ACCEPTED BY IMPLEMENTATIONS FOR WHICH Boolean'Size = 1.)

THE BIT RANGE FOR Data_Address EXTENDS PAST THE LAST BIT OF STORAGE UNIT (BYTE) 1 ITSELF, TO THE 23RD BIT AFTER THE FIRST ONE. THE IMPLEMENTATION MIGHT ALLOW ALL BIT RANGES TO BE NUMBERED FROM THE START OF BYTE 0, IN WHICH CASE THE BITS COULD BE NUMBERED 0 TO 63 AS ON THE PREVIOUS SLIDE. FOR EXAMPLE, Byte_Count COULD BE DESCRIBED AS "AT 0 RANGE 48 .. 63."

EXAMPLE OF RECORD REPRESENTATION (CONTINUED)

- RECORD REPRESENTATION CLAUSE:

```
for CCW_Type use
   record at mod 8;
      Command_Code                           at 0 range 0 .. 7;
      Date_Address                           at 1 range 0 .. 23;
      Chain_Data_Flag                        at 4 range 0 .. 0;
      Chain_Command_Flag                     at 4 range 1 .. 1;
      Suppress_Length_Indication_Flag        at 4 range 2 .. 2;
      Skip_Flag                              at 4 range 3 .. 3;
      Program_Controlled_Interrupt_Flag      at 4 range 4 .. 4;
      Indirect_Address_Flag                  at 4 range 5 .. 5;
      Byte_Count                             at 6 range 0 .. 15;
   end record;
```

- SHOULD BE FOLLOWED BY A LENGTH CLAUSE TO GUARANTEE NO PADDING BITS AT END:

```
for CCW_Type'Size use 64;
```

- NOW REST OF PROGRAM CAN USE CCW_Type AS AN ORDINARY RECORD TYPE, WITHOUT REGARD TO THE PLACEMENT OF INDIVIDUAL COMPONENTS.

INSTRUCTOR NOTES

THE EXPRESSION OF TYPE System.Address IN AN ADDRESS CLAUSE, EVEN IF IT IS ONLY AN INTEGER LITERAL, IS CONSIDERED AN IMPLIED USE OF THE PACKAGE SYSTEM. THAT'S WHY THE with CLAUSE IS REQUIRED.

THE USE OF ADDRESS CLAUSES FOR HARDWARE INTERRUPTS WILL BE SKETCHED BRIEFLY IN THE OVERVIEW OF TASKING.

USE OF ADDRESS CLAUSES TO ACHIEVE OVERLAYS MAKES A PROGRAM "ERRONEOUS." THAT MEANS THE PROGRAM IS OFFICIALLY ILLEGAL BUT THERE MAY BE NO WAY TO DETECT THIS.

SPECIFYING THE ADDRESS OF AN ENTITY

- ADDRESS CLAUSE:

  for [entity name] use at [expression of type System.Address];

- REQUIRES A with CLAUSE FOR PACKAGE SYSTEM.

- CAN BE USED TO SPECIFY THE ADDRESS OF:

  - AN OBJECT (STARTING ADDRESS OF DATA)
  - A PROGRAM UNIT (ADDRESS OF FIRST MACHINE INSTRUCTION)

- APPROPRIATE USES:

  - WHEN THE HARDWARE USES A SPECIFIC LOCATION FOR CERTAIN DATA (E.G. A PROGRAM STATUS WORD OR AN INTERRUPT VECTOR). ONCE THE ADDRESS IS SPECIFIED, THE REST OF THE PROGRAM CAN USE THE DATA AS AN ORDINARY VARIABLE OR CONSTANT.
  - WHEN THE HARDWARE BRANCHES TO A CERTAIN ADDRESS IN SPECIAL CIRCUMSTANCES (E.G. POWER-UP OR INTERRUPTS). SUBPROGRAMS TO HANDLE THESE CIRCUMSTANCES CAN BE PLACED IN THE REQUIRED LOCATIONS.

- ADDITIONAL USE:

  THERE IS A WAY TO USE ADDRESS CLAUSES IN CONJUNCTION WITH TASKS TO PROVIDE AN ABSTRACT VIEW OF HARDWARE INTERRUPTS IN TERMS OF INTER-TASK COMMUNICATION.

- FORBIDDEN USE:

  ADDRESS CLAUSES ARE NOT TO BE USED TO CAUSE OVERLAYS OF OBJECTS OR PROGRAM UNITS.

INSTRUCTOR NOTES

TO REVIEW, A REPRESENTATION CLAUSE (OR THE PACK PRAGMA) MAY GO IN THE SEQUENCE OF DECLARATIONS IN A PACKAGE SPECIFICATION OR THE DECLARATIVE PART OF A BLOCK STATEMENT OR UNIT BODY, FOLLOWING THE DECLARATION OF THE TYPE IT AFFECTS BUT PRECEDING ANY USE OF THAT TYPE REQUIRING KNOWLEDGE OF ITS REPRESENTATION.

THOUGH AN IMPLEMENTATION MAY IGNORE ANOTHER IMPLEMENTATION'S PRAGMAS THERE IS A DANGER THAT IT WILL MISTAKENLY RECOGNIZE IT AS ONE OF ITS OWN IMPLEMENTATION-DEFINED PRAGMAS THAT HAS THE SAME NAME BUT A DIFFERENT MEANING.

REPRESENTATION PRAGMAS

- THE PACK PRAGMA
  - FORM:

    pragma Pack (composite type name);

  - MEANING:

    COMPILER SHOULD MINIMIZE THE GAPS BETWEEN COMPONENTS (EVEN IF THIS ELIMINATES THE ALIGNMENT NEEDED FOR EFFICIENT ACCESS).

  - PLACEMENT:

    SAME AS FOR A REPRESENTATION CLAUSE FOR THE TYPE

- AN IMPLEMENTATION MAY DEFINE ADDITIONAL REPRESENTATION PRAGMAS.
  - PRAGMAS PLACED IN A PROGRAM FOR THE BENEFIT OF ANOTHER IMPLEMENTATION, BUT NOT RECOGNIZED BY YOUR IMPLEMENTATION, MUST BE IGNORED BY YOUR IMPLEMENTATION.

INSTRUCTOR NOTES

COMPACT REPRESENTATIONS MAY REQUIRED MASKING AND SHIFTING INSTRUCTIONS BEFORE USE CAN BE MADE OF A COMPONENT OCCUPYING PART OF A STORAGE UNIT.

TYPE CONVERSION IS ALWAYS DEFINED BETWEEN A DERIVED TYPE AND ITS PARENT TYPE.

TYPES AND REPRESENTATIONS

- AT MOST ONE REPRESENTATION MAY BE SPECIFIED FOR EACH TYPE.
- SOMETIMES IT IS DESIRABLE TO HAVE DIFFERENT WAYS OF REPRESENTING THE SAME INFORMATION.
  - ONE WAY FOR FAST ACCESS (EACH COMPONENT ALIGNED IN AN ADDRESSABLE STORAGE BOUNDARY)
  - ONE WAY FOR COMPACTNESS OR COMPATIBILITY WITH AN EXTERNALLY SPECIFIED FORMAT (DIFFERENT COMPONENTS USING DIFFERENT BITS OF THE SAME STORAGE UNIT)
- SOLUTION: DERIVED TYPES

```
type Format_1 is ...;
type Format_2 is new Format_1;

for Format_1 use ...;
for Format_2 use ...;
```

- THEN TYPE CONVERSION CAUSES A CHANGE IN REPRESENTATION:

```
F1 : Format_1;
F2 : Format_2;
      .
      .
      .
F1 := Format_1(F2);
F2 := Format_2(F1);
```

INSTRUCTOR NOTES

DON'T GET INVOLVED WITH FLOATING POINT AND TWO'S COMPLEMENT FORMATS. THE POINT IS SIMPLY THAT BITS IN A GIVEN POSITION MAY HAVE COMPLETELY UNRELATED MEANINGS IN THE INTERNAL REPRESENTATIONS OF TYPES FLOAT AND INTEGER.

THE INCONSISTENCIES IN THE LAST BULLET ARE ORDERED FROM VERY LOW-LEVEL INCONSISTENCIES TO VERY HIGH-LEVEL INCONSISTENCIES.

UNCHECKED CONVERSION

- ALLOWS A BIT REPRESENTATION FOR A VALUE IN ONE TYPE TO BE INTERPRETED AS THE BIT REPRESENTATION FOR A VALUE IN ANOTHER TYPE.

- NOT THE SAME AS ORDINARY TYPE CONVERSION, IN WHICH THE CONVERTED VALUE IS RELATED IN SOME ABSTRACT SENSE TO THE ORIGINAL VALUE.

  EXAMPLE: Float_to_Integer UNCHECKED CONVERSION, ASSUMING 32-BIT IBM FORMAT REPRESENTATIONS OF FLOAT AND INTEGER.

  ± EXPONENT MANTISSA

0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

  FLOAT VALUE 65.25

  ⇓ REINTERPRETATION OF SAME BIT PATTERN

  TWO'S COMPLEMENT BINARY INTEGER

0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

  INTEGER VALUE 1_076_106_240

- CAN PRODUCE INCONSISTENT VALUES OF THE TARGET TYPE.
  - UNNORMALIZED FLOATING POINT VALUES
  - ACCESS VALUES POINTING "NOWHERE"
  - MEANINGLESS CODES FOR ENUMERATION TYPE VALUES
  - VIOLATIONS OF CONSTRAINTS
  - IMPROPER MANIPULATION OF PRIVATE TYPES, LEADING TO INCONSISTENT DATA STRUCTURES
  - ETC., ETC., ETC.

  (THAT'S WHY ITS CALLED "UNCHECKED")

INSTRUCTOR NOTES

THE GENERIC FUNCTION Unchecked_Conversion IS ANALOGOUS IN NAME, FORM, AND SPIRIT TO THE GENERIC PROCEDURE Unchecked_Deallocation GIVEN EARLIER.

COMING SLIDES CONTAIN EXAMPLES OF THE USE OF THIS FUNCTION.

HOW TO PERFORM UNCHECKED CONVERSION

- NAME THE PREDEFINED LIBRARY GENERIC FUNCTION Unchecked_Conversion IN A with CLAUSE.

GENERIC DECLARATION:

```
generic
   type Source is limited private;
   type Target is limited private;
function Unchecked_Conversion (S: Source) return Target;
```

- INSTANTIATE THE GENERIC FUNCTION.
  - IMPLEMENTATION MAY IMPOSE RESTRICTIONS ON ALLOWABLE DESTINATIONS, SUCH AS REQUIRING THAT Source AND Target TYPES HAVE THE SAME LENGTH. (SEE APPENDIX F OF YOUR IMPLEMENTATION'S REFERENCE MANUAL.)

- CALL THE INSTANCE.

INSTRUCTOR NOTES

EXAMPLES OF THIS KIND OF USE ARE FOUND IN THE EXTENDED LOW-LEVEL PROGRAMMING EXAMPLE.

A QUITE DIFFERENT USE OF UNCHECKED CONVERSION IS SHOWN ON THE NEXT SLIDE.

WHEN TO PERFORM UNCHECKED CONVERSION

- ALMOST NEVER -- APPROPRIATE USES ARE VERY RARE.

- OCCASIONALLY, WHEN INTERFACING WITH HARDWARE OR ASSEMBLY-LEVEL SOFTWARE, IT IS NECESSARY TO VIEW THE SAME SEQUENCE OF BITS AS REPRESENTING, FOR EXAMPLE, AN INTEGER, A CHARACTER, AND A SEQUENCE OF FLAGS.
  - DIFFERENT OPERATIONS ARE APPLICABLE FOR EACH VIEW.
  - ADA'S STRONG TYPING RULES REQUIRE A DIFFERENT TYPE FOR EACH VIEW.

- INTERNALLY, A CALL ON AN INSTANCE OF Unchecked_Conversion DOES NOTHING. THE BITS REPRESENTING THE PARAMETER ARE SIMPLY RETURNED AS THE BITS REPRESENTING THE RESULT.

- EXTERNALLY, THE CALL CONVERTS A VALUE IN ONE TYPE TO A VALUE IN ANOTHER TYPE.

STATEMENT-BY-STATEMENT EXPLANATION OF Tree_Hash_Value:

STATEMENT 1 - ASSIGNS Integer_Value A UNIQUE INTEGER FOR EACH POSSIBLE VALUE OF THE ACCESS VARIABLE TREE.

STATEMENT 2 - IF ALL TREE NODES ARE THE SAME SIZE AND THEY ARE ALLOCATED FROM CONSECUTIVE STORAGE UNITS, AND IF ACCESS VALUES ARE REPRESENTED INTERNALLY BY MACHINE ADDRESSES, THESE INTEGER VALUES WILL BE EVENLY SPACED. THIS WILL RESULT IN AN UNEVENLY DISTRIBUTED HASH FUNCTION. DIVIDING BY THE PRESUMED INTERNAL BETWEEN INTEGER VALUES WOULD MAP TREE VALUES ONTO CONSECUTIVE INTEGERS, PRODUCING A BETTER HASH FUNCTION.

STATEMENT 3 - ON SOME MACHINES, ADDRESSES ARE TREATED AS UNSIGNED VALUES (IN WHICH THE HIGH ORDER BIT IS PART OF THE MAGNITUDE) AND INTEGERS ARE TREATED AS SIGNED VALUES (IN WHICH THE HIGH ORDER BIT INDICATES THE SIGN). THUS THE BITS REPRESENTING AN ADDRESS MAY REPRESENT A NEGATIVE VALUE WHEN INTERPRETED AS AN INTEGER. THE APPLICATION OF abs ELIMINATES THE POSSIBILITY OF NEGATIVE VALUES AND THE ADDITION OF 1 ELIMINATES THE POSSIBILITY OF ZERO VALUES. THUS THE RETURNED VALUE IS OF SUBTYPE POSITIVE, AS REQUIRED BY THE GENERIC PARAMETER DECLARATION OF Hash_Value IN Lookup_Table_Package.

USE OF UNCHECKED CONVERSION FOR HASHING

- A LEGITIMATE HIGH-LEVEL USE OF Unchecked_Conversion IS IN A HASHING FUNCTION, WHEN THERE IS NO ABSTRACT WAY (LIKE Character'Pos) TO EXTRACT A NUMERIC VALUE FROM THE KEY TO BE HASHED.

- SUPPOSE THE PACKAGE Tree_Package DEFINES THE TYPE Tree_Type AS AN ACCESS TYPE. THE PACKAGE ALSO PROVIDES A NAMED NUMBER Storage_Units_Per_Node GIVING THE NUMBER OF STORAGE UNITS ALLOCATED FOR EACH TREE NODE.

- THE FOLLOWING FUNCTION RETURNS A DIFFERENT VALUE OF SUBTYPE POSITIVE FOR EACH VALUE OF TYPE Tree_Type:

```
with Unchecked_Conversion, Tree_Package;
use Tree_Package;

function Tree_Hash_Value (Tree : Tree_Type) return Positive is
   function Pointer_Bits_As_Integer is new
      Unchecked_Conversion (Source => Tree_Type, Target => Integer);
   -- to convert value of access type Tree_Type to value of
   -- type Integer (no predefined numeric type conversion would
   -- work here)
   Integer_Value, Adjusted_Value : Integer;
begin -- Tree_Hash_Value
   Integer_Value := Pointer_Bits_As_Integer (Tree);   -- use point as number
   Adjusted_Value := Integer_Value/Storage_Units_Per_Node;  -- apply a simple hashing
                                                            -- algorithm to it
   return abs Adjusted_Value + 1;  -- when interpreted as a number, the access value
                                   -- could have been negative
end Tree_Hash_Value;
```

INSTRUCTOR NOTES

USE OF UNCHECKED CONVERSION FOR HASHING (CONTINUED)

- THIS FUNCTION CAN BE USED IN INSTANTIATING THE HASH TABLE PACKAGE PRESENTED EARLIER:

```
with Lookup_Table_Package, Tree_Package, Tree_Hash_Value;

package Tree_Table_Package is new
   Lookup_Table_Package
      (Key_Type   => Tree_Package.Tree_Type,
       Data_Type  => Tree_Package.Tree_Type,
       Null_Data  => null,
       Hash_Value => Tree_Hash_Value);
```

INSTRUCTOR NOTES

THIS IS A PHILOSOPHICAL ASIDE DESIGNED TO (1) PROVIDE A BREAK FROM THE INTENSE PACE OF THIS SECTION AND (2) GET STUDENTS THINKING ABOUT THE NATURE OF "PORTABILITY." THERE ARE SEVERAL WAYS TO ANSWER THE QUESTION AT THE TOP OF THE SLIDE.

IS THIS HASH FUNCTION PORTABLE?

- NO:

  THE VALUE IT RETURNS DEPENDS DIRECTLY ON THE INTERNAL REPRESENTATION OF ACCESS VALUES AND THE IMPLEMENTATION'S SCHEME FOR CREATING ALLOCATED VARIABLES.

- YES:

  WE USE A HASH FUNCTION AS A RANDOMIZED MAPPING FROM TREE VALUES TO ARBITRARY INTEGERS, AND WE DON'T REALLY CARE ABOUT THE EXACT VALUES. (IN FACT, ONE IMPLEMENTATION MIGHT PRODUCE DIFFERENT HIGH VALUES FOR DIFFERENT RUNS OF THE SAME PROGRAM, DEPENDING ON WHERE THE PROGRAM IS LOADED INTO MEMORY.)

  FROM AN ABSTRACT POINT OF VIEW, THE HASH TABLE WILL PERFORM THE SAME FUNCTION REGARDLESS OF HOW TREES ARE HASHED.

- SORT OF:

  SUBTLE CHARACTERISTICS OF THE ACCESS VALUE REPRESENTATION OR THE ALLOCATION SCHEME COULD AFFECT THE EXPECTED NUMBER OF HASH TABLE COLLISIONS. THUS THE HASH TABLE PERFORMANCE MIGHT BE MUCH WORSE FOR ONE IMPLEMENTATION EVEN THOUGH THE ULTIMATE ABSTRACT EFFECT IS THE SAME.

INSTRUCTOR NOTES

TYPICALLY, THIS PRAGMA LEADS TO FASTER BUT LARGER PROGRAMS.

FOR BULLET THREE, EXPLAIN THE DISTINCTION BETWEEN NUMBER OF <u>PLACES</u> A SUBPROGRAM IS CALLED FROM AND NUMBER OF <u>TIMES</u> IT IS CALLED.

IF A SUBPROGRAM NAME GIVEN IN THE PRAGMA IS OVERLOADED, IT APPLIES TO ALL VERSIONS FOR WHICH THE PRAGMA'S PLACEMENT IS LEGAL.

THE INLINE PRAGMA

- SPECIFIES THAT ALL CALLS ON A SUBPROGRAM SHOULD GENERATE COPIES OF THE TRANSLATION OF THE BODY ITSELF RATHER THAN JUMPS TO THE TRANSLATOR OF THE BODY.

- A COMPILER MAY IGNORE THIS ADVICE ON A CALL-BY-CALL BASIS. (FOR RECURSIVE SUBPROGRAMS, IT HAS TO IGNORE IT AT LEAST PARTIALLY.)

- APPROPRIATE FOR SHORT SUBPROGRAMS CALLED MANY TIMES (BUT FROM FEW PLACES).

- FORM: pragma Inline (subprogram name {, subprogram name });

- PLACEMENT: IN THE SAME DECLARATIVE PART OR PACKAGE SPECIFICATION AS THE SUBPROGRAM DECLARATION (OR SUBPROGRAM BODY IF THERE IS NO DECLARATION); OR IMMEDIATELY AFTER A SEPARATELY COMPILED SUBPROGRAM DECLARATIVE (OR SUBPROGRAM BODY IF THERE IS NO DECLARATIVE).

  - GOOD STYLE FOR SUBPROGRAMS DECLARED IN THE VISIBLE PART OF THE PACKAGE SPECIFICATION IS TO PLACE THE PROGRAM IN THE PRIVATE PART (EVEN IF THE PACKAGE HAS NO PRIVATE TYPES).

  - THE PRAGMA IS CONCERNED WITH THE IMPLEMENTATION OF THE SUBPROGRAM RATHER THAN ITS EXTERNAL INTERFACE.

- NORMALLY, RECOMPILATION OF A SUBPROGRAM BODY WITHOUT RECOMPILATION OF ITS DECLARATION DOES NOT REQUIRE RECOMPILATION OF OTHER PROGRAM UNITS CALLING THE SUBPROGRAM. IF THE INLINE PRAGMA IS USED, RECOMPILATON MAY BE REQUIRED AFTER ALL.

INSTRUCTOR NOTES

FOR SEQUENCES OF MORE THAN A FEW MACHINE INSTRUCTIONS, THE INSTRUCTIONS SHOULD BE WRITTEN IN ASSEMBLY LANGUAGE AND INCORPORATED IN THE ADA PROGRAM USING THE INTERFACE PRAGMA.

INTERFACE WITH OTHER LANGUAGES

- FOR SHORT SEQUENCES OF SPECIFIC MACHINE-LANGUAGE INSTRUCTIONS:

  CODE PROCEDURES

- FOR CALLING SUBPROGRAMS WRITTEN IN SOME OTHER LANGUAGE:

  THE INTERFACE PRAGMA

- BOTH ARE HIGHLY IMPLEMENTATION-DEPENDENT. SEE APPENDIX F OF YOUR IMPLEMENTATION'S REFERENCE MANUAL.

TYPICALLY, THE OP-CODE COMPONENT OF A MACHINE INSTRUCTION RECORD TYPE WILL BE A DISCRIMINANT GOVERNING A VARIANT. THEN THE NUMBER AND TYPES OF OTHER COMPONENTS WILL DEPEND ON THE OP-CODE.

AN IMPLEMENTATION MAY PROVIDE PRAGMAS SPECIFYING CALLING CONVENTIONS FOR CODE PROCEDURES (OR OTHER PROCEDURES). THESE COULD DETERMINE, FOR EXAMPLE, WHICH REGISTERS CONTAIN OR POINT TO PARAMETERS, RETURN ADDRESSES, ETC.

IF AN IMPLEMENTATION FORBIDS PARAMETERS FOR CODE PROCEDURES, A CODE PROCEDURE CAN BE NESTED IN AN ORDINARY PROCEDURE WITH PARAMETERS. THEN THE CODE PROCEDURE CAN REFER TO GLOBAL VARIABLES DECLARED IN THE ORDINARY PROCEDURE, AS SHOWN IN THE HYPOTHETICAL EXAMPLE.

THE EXAMPLE ASSUMES THAT THE SVC, OR SUPERVISOR CALL, INSTRUCTION INVOKES OPERATING SYSTEM ROUTINES. SUPERVISOR CALL 4A HEXADECIMAL SENDS A MESSAGE TO THE OPERATOR CONSOLE, ASSUMING THAT REGISTER R0 CONTAINS THE NUMBER OF CHARACTERS IN THE MESSAGE AND REGISTER R1 CONTAINS THE ADDRESS OF THE FIRST CHARACTER.

THE IDENTIFIERS LOAD, SVC, R0, AND R1 ARE PRESUMED TO BE DECLARED IN THE PACKAGE Machine_Code. THE OUTER PROCEDURE REFERS TO Message(Message'First)'Address RATHER THAN Message'Address BECAUSE THE INTERNAL REPRESENTATION OF A STRING MIGHT BEGIN WITH A LENGTH CODE RATHER THAN WITH THE FIRST CHARACTER.

THE EXPRESSIONS IN THE MACHINE INSTRUCTION AGGREGATES SHOULD EVALUATE TO WHAT GOES IN THE INSTRUCTION -- TYPICALLY THE ADDRESS OF THE OPERAND RATHER THAN THE VALUE OF THE OPERAND.

CODE PROCEDURES

- AN IMPLEMENTATION MAY (BUT NEED NOT) PROVIDE A PACKAGE NAMED Machine_Code, PROVIDING ONE OR MORE RECORD TYPES WHOSE VALUES CORRESPOND TO MACHINE LANGUAGE INSTRUCTIONS.

- IN A CODE PROCEDURE, EACH STATEMENT CONSISTS OF A QUALIFIED RECORD AGGREGATE OF THIS TYPE.

- RESTRICTIONS ON CODE PROCEDURES:
  - DECLARATIVE PART MAY ONLY CONTAIN use CLAUSES.
  - NO EXCEPTION HANDLERS
  - AN IMPLEMENTATION MAY PROVIDE FURTHER RESTRICTIONS, SUCH AS REQUIRING ALL EXPRESSIONS IN THE AGGREGATES TO BE STATIC, OR PROHIBITING PARAMETERS.

- THE PACKAGE Machine_Code MUST BE MADE AVAILABLE BY A <u>with</u> CLAUSE.

INSTRUCTOR NOTES

- HYPOTHETICAL EXAMPLE:

```
with Machine_Code, System; use Machine_Code;

procedure Send_Message_To_Operator (Message : in String) is

   Character_Count  : Natural := Message'Length;
   Starting_Address : System.Address := Message (Message'First)'Address;

   procedure Send_Message_SVC is
   begin -- Send_Message_SVC
      Machine_Instruction'(LOAD, R0, Character_Count'Address);
      Machine_Instruction'(LOAD, R1, Starting_Address'Address);
      Machine_Instruction'(SVC, 16#4A#);
   end Send_Message_SVC;

   pragma Inline (Send_Message_SVC);

begin -- Send_Message_To_Operator

   Send_Message_SVC;

      .
      .
      .

end Send_Message_To_Operator;
```

INSTRUCTOR NOTES

IF THE SUBPROGRAM NAME IS OVERLOADED, IT REFERS TO ALL VERSIONS DECLARED EARLIER IN THE SAME SEQUENCE OF DECLARATIONS OR COMPILATION UNITS.

THE INTERFACE PRAGMA

- ALLOWS ADA PROGRAMS TO CALL SUBPROGRAMS WRITTEN IN ANOTHER LANGUAGE (NOT VICE VERSA).
- FORM: `pragma interface ( language name , subprogram name );`
- PLACEMENT:
  - IN A DECLARATIVE PART OF A BLOCK STATEMENT, SUBPROGRAM BODY, PACKAGE BODY, OR TASK BODY, PROVIDED THAT A DECLARATION FOR THE SUBPROGRAM OCCURS EARLIER IN THE SAME DECLARATIVE PART.
  - IN THE VISIBLE OR PRIVATE PART OF A PACKAGE SPECIFICATION, PROVIDED THAT A DECLARATION FOR THE SUBPROGRAM OCCURS EARLIER IN THE PACKAGE SPECIFICATION.
  - AFTER A SEPARATELY COMPILED DECLARATION OF THE SUBPROGRAM, BUT BEFORE ANY OTHER COMPILATION UNIT.
- THE ADA COMPILER MUST NOT BE GIVEN A BODY FOR THE SUBPROGRAM, SINCE THE BODY IS WRITTEN IN ANOTHER LANGUAGE.
- AN IMPLEMENTATION MAY RESTRICT THE ALLOWABLE SUBPROGRAMS, E.G. REQUIRING PARAMETERS TO BELONG TO CERTAIN PREDEFINED TYPES.
- THE IMPLEMENTATION IS RESPONSIBLE FOR SATISFYING THE CALLING CONVENTIONS OF THE OTHER LANGUAGE.

INSTRUCTOR NOTES

A SPECIFIC EXAMPLE FOLLOWS AS PART OF AN EXTENDED EXAMPLE OF LOW-LEVEL PROGRAMMING.

DEVICE-LEVEL INPUT/OUTPUT

- PREDEFINED PACKAGE NAMED Low_Level_IO
  - PROVIDES DATA TYPES FOR DEVICES AND DATA
  - PROVIDES OVERLOADED VERSIONS OF TWO PROCEDURES:

```
procedure Send_Control
   (Device : in some type for devices ;
    Data   : in out some type for data );

procedure Receive_Control
   (Device : in some type for devices ;
    Data   : in out some type for data );
```

- MEANING OF THESE PROCEDURES DEPENDS ON THE IMPLEMENTATION

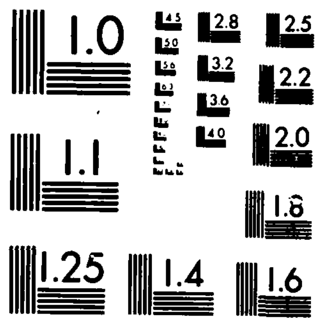
- USE OF Low_Level_IO IS CONSIDERED HIGHER-LEVEL THAN USE OF CODE PROCEDURES.

- Low_Level_IO MUST BE NAMED IN A with CLAUSE.

INSTRUCTOR NOTES

AD-A165 077 ADA (TRADEMARK) TRAINING CURRICULUM: ADVANCED ADA TOPICS L305 TEACHER'S GUIDE VOLUME 3(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K506 2/2

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

SECTION 22

EXAMPLE OF LOW LEVEL PROGRAMMING

INSTRUCTOR NOTES

THIS SECTION FOLLOWS THESE STEPS IN TURN. WE BEGIN WITH A HYPOTHETICAL VERSION OF Low_Level_IO, FOLLOWED BY A DESCRIPTION OF THE HARDWARE, FOLLOWED BY THE SPECIFICATION AND BODY OF THE INTERFACE PACKAGE.

LOW-LEVEL PROGRAMMING EXAMPLE

- APPLICATION: AN ANTENNA-TUNER-INTERFACE.

- MUST ESTABLISH GROUND RULES FIRST
  - A HYPOTHETICAL VERSION OF Low_Level_IO
  - HARDWARE SPECIFICATION

- NEXT STEP IS TO DESIGN THE SOFTWARE INTERFACE.

- FINAL STEP IS TO IMPLEMENT THE SOFTWARE INTERFACE.

INSTRUCTOR NOTES

EMPHASIZE THAT THIS IS A <u>HYPOTHETICAL</u> RENDITION OF ONE IMPLEMENTATION'S VERSION OF Low_Level_IO.

THE TARGET MACHINE IS AN INTEL 8080-BASED PROCESSOR. THERE ARE TWO I/O INSTRUCTIONS -- IN AND OUT. IN SENDS AN 8-BIT BYTE OF DATA TO A SPECIFIED CHANNEL AND OUT RECEIVES A BYTE OF DATA FROM A SPECIFIED CHANNEL. CHANNELS ARE NUMBERED 0 TO 255.

THE PROCEDURES Send_Control AND Receive_Control ARE DEFINED TO CORRESPOND TO OUT AND IN INSTRUCTIONS, RESPECTIVELY.

THE TYPE DECLARATIONS PROVIDE THREE HIGH-LEVEL VIEWS OF A BYTE OF DATA -- AS AN UNSIGNED INTEGER, AS A SEQUENCE OF BITS, AND AS A CHARACTER. THESE ARE OVERLOADED VERSIONS OF Send_Control AND Receive_Control CORRESPONDING TO EACH OF THESE VIEWS.

A HYPOTHETICAL VERSION OF Low_Level_IO

```
package Low_Level_IO is

   type Channel_Number is range 0 .. 255;
   subtype Integer_Byte is Integer range 0 .. 255;
   type Bit_Sequence_Byte is array (0 .. 7) of Boolean;
      -- Conventionally, bits in this machine are numbered
      -- right to left.  If B is of type Bit_Sequence_Byte,
      -- B(i) is bit 7-i of the byte.
   subtype Character_Byte is Character;

   procedure Send_Control
      (Device: in Channel_Number; Data: in out Integer_Byte);

   procedure Send_Control
      (Device: in Channel_Number; Data: in out Bit_Sequence_Byte);

   procedure Send_Control
      (Device: in Channel_Number; Data: in out Character_Byte);

   procedure Receive_Control
      (Device: in Channel_Number; Data: in out Integer_Byte);

   procedure Receive_Control
     (Device: in Channel_Number; Data: in out Bit_Sequence_Byte);

   procedure Receive_Control
     (Device: in Channel_Number; Data: in out Character_Byte);

end Low_Level_IO;
```

INSTRUCTOR NOTES

THE NEXT THREE SLIDES DESCRIBE THE TUNER CONTROL INTERFACE, TUNER STATUS INTERFACE, AND TAP POSITION INDICATOR IN DETAIL.

DESCRIPTION OF THE ANTENNA TUNER

- AN ANTENNA CAN BE SET FOR EITHER THE 2-11 MHZ BAND OR THE 11-30 MHZ BAND.
- AN "ANTENNA TAP," DRIVEN BY A MOTOR, THAT CAN BE MOVED UP OR DOWN TO TUNE THE ANTENNA TO DIFFERENT FREQUENCIES WITHIN A BAND.
- A SWITCH TO ALLOW RADIO FREQUENCY (RF) ENERGY TO PASS TO THE ANTENNA, PLUS A SENSOR INDICTING WHETHER IT IS ARRIVING THERE.
- INTERFACE HAS THREE COMPONENTS:
  - TUNER CONTROL INTERFACE

    SELECTS A BAND, DRIVES THE MOTOR, TURNS RF OFF AND ON
  - TUNER STATUS INTERFACE

    INDICATES WHETHER RF ENERGY IS ARRIVING AT THE ANTENNA, WHETHER THE TAP IS AT ITS TOPMOST OR BOTTOMMOST POSITION.
  - TAP POSITION INDICATOR

    REPORTS THE CURRENT POSITION OF THE TAP ON A SCALE FROM 0 TO 255.

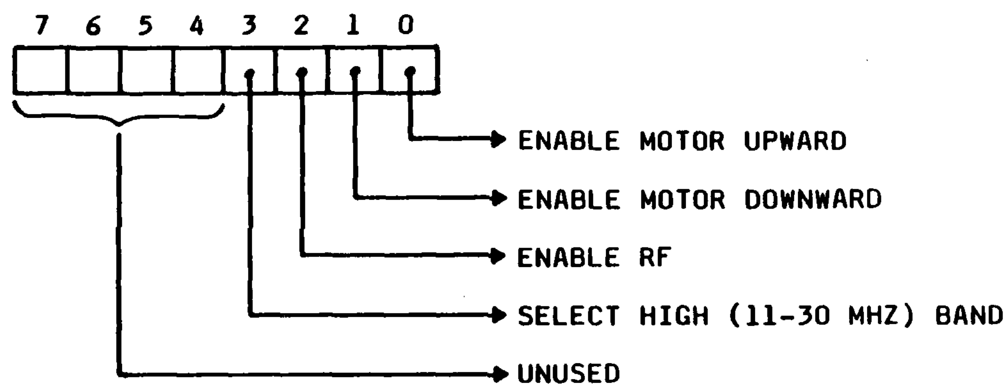
INSTRUCTOR NOTES

NOTICE THAT BITS ARE NUMBERED LEFT-TO-RIGHT.

IT IS THE RESPONSIBILITY OF THE SOFTWARE TO ENSURE THAT THE MOTOR IS NOT ENABLED IN BOTH DIRECTIONS AT ONCE.

TUNER CONTROL INTERFACE HARDWARE SPECIFICATION

- THE TUNER CAN BE CONTROLLED BY SENDING THE FOLLOWING BYTE TO CHANNEL 10 HEXADECIMAL:



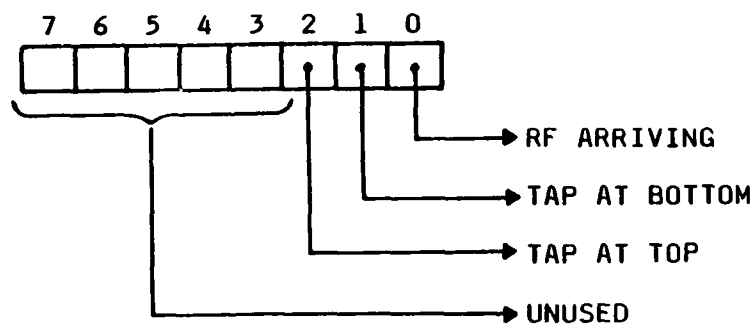
- MOTOR DAMAGE CAN OCCUR IF BITS 1 AND 0 ARE ON SIMULTANEOUSLY. (THIS IS CONSIDERED UNDESIRABLE.)

INSTRUCTOR NOTES

CHANNEL 16#10# ACTS AS BOTH AN OUTPUT CHANNEL FOR THE TUNER CONTROL INTERFACE AND AN INPUT CHANNEL FOR THE TUNER STATUS INTERFACE.

TUNER STATUS INTERFACE HARDWARE SPECIFICATION

- TUNER STATUS CAN BE DETERMINED BY RECEIVING THE FOLLOWING BYTE FROM CHANNEL 10 HEXADECIMAL:



INSTRUCTOR NOTES

THIS IS WHERE THE VERSION OF Receive_Control WITH A DATA PARAMETER OF TYPE Integer_Byte COMES IN HANDY.

TAP POSITION INDICTOR HARDWARE SPECIFICATION

- TAP POSITION CAN BE OBTAINED BY RECEIVING A BYTE FROM CHANNEL 20 HEXADECIMAL.
- THE BYTE SHOULD BE INTERPRETED DIRECTLY AS AN INTEGER BETWEEN 0 AND 255.

INSTRUCTOR NOTES

THE SOFTWARE INTERFACE IS BASICALLY A TRANSLATION OF THE HARDWARE INTERFACE INTO HIGH-LEVEL ADA TERMS. IT SHOULD ALLOW AN ADA PROGRAMMER TO USE THE DEVICE WITHOUT EVER SEEING THE BIT-BY-BIT HARDWARE SPECIFICATIONS.

THE INTERFACE MODULE IS SEEN AS A TOOL TO BE USED IN WRITING A HIGHER-LEVEL MODULE THAT WILL PROVIDE ABSTRACT OPERATIONS BASED ON THE PROBLEM TO BE SOLVED. THE INTERFACE MODULE ITSELF IS BASED ON THE ABSTRACT STRUCTURE OF THE DEVICE.

CONTROLLING THE MOTOR, ENABLING RF FLOW, AND SELECTING A FREQUENCY BAND ARE THREE RATHER UNRELATED OPERATIONS. OUR FIRST INSTINCT WOULD BE TO PROVIDE THREE SEPARATE PROCEDURES. HOWEVER, EACH PROCEDURE CALL WOULD THEN REQUIRE A SEPARATE I/O OPERATION. IF THE APPROPRIATE SETTINGS FOR THE MOTOR, RF FLOW, AND FREQUENCY BAND ARE COMPUTED ON A FERIODIC BASIS, SAY EVERY 10 MILLISECONDS, THIS WOULD REQUIRE THREE OUTPUT OPERATIONS EVERY 10 MILLISECONDS INSTEAD OF ONE. (ON THE OTHER HAND, SEPARATE PROCEDURES MAY BE APPROPRIATE AT A HIGHER LEVEL.)

SOFTWARE INTERFACE

- REQUIREMENTS
  - A PROCEDURE TO CONTROL THE MOTOR, THE RF SWITCH, AND THE FREQUENCY BAND SELECTION.
  - A PROCEDURE TO DETERMINE WHETHER RF ENERGY IS ARRIVING AT THE ANTENNA AND WHETHER THE TAP IS AT AN END POINT.
  - A FUNCTION RETURNING THE CURRENT TAP POSITION.
  - TYPES FOR MOTOR, RF, AND BAND SETTINGS: TYPE FOR TAP POSITIONS.
- RATIONALE
  - PROVIDE A "VIRTUAL DEVICE" THAT CAN BE OPERATED DIRECTLY IN TERMS OF HIGH-LEVEL ADA FEATURES -- SUBPROGRAMS, ENUMERATION TYPES, AND INTEGER TYPES.
  - GIVE THE USER FULL FLEXIBILITY TO OPERATE THE DEVICE IN EVERY MEANINGFUL WAY.
    - HE SHOULD BE ABLE TO TURN ON THE MOTOR AND SELECT A FREQUENCY BAND WITH A SINGLE I/O INSTRUCTION
    - HE SHOULD NOT BE ABLE TO SET THE MOTOR-UPWARD AND MOTOR-DOWNWARD BITS TO 1 SIMULTANEOUSLY.
  - ALLOW FOR THE DEFINITION OF HIGHER-LEVEL, MORE ABSTRACT, MORE RESTRICTED OPERATIONS USING THE DEVICE.

INSTRUCTOR NOTES

THE MOTOR-UPWARD AND MOTOR-DOWNWARD BITS ARE VIEWED AS TOGETHER FORMING A 2-BIT DESCRIPTION OF THE MOTOR SETTING. (WHAT KIND OF TWO-BIT TYPE IS THIS ANYWAY?) ONE OF THE FOUR POSSIBLE SETTINGS FOR THESE BITS IS IMPROPER, SO THE LEGAL COMBINATIONS ARE USED AS THE ENCODING OF A THREE-VALUE ENUMERATION TYPE. THE ENUMERATION REPRESENTATION CLAUSE FOR MOTOR-SETTING TYPE USES BINARY INTEGER LITERALS TO EMPHASIZE THE RELATIONSHIP OF THE ENCODING TO THE TWO BITS.

A SIMILAR APPROACH IS TAKEN WITH THE TAP-AT-TOP AND TAP-AT-BOTTOM BITS IN Tap_Extremity_Type.

THE LENGTH CLAUSES ENSURE REPRESENTATIONS THAT DO NOT CONTAIN PADDING BITS.

A USER OF THE PACKAGE NEED NOT BE CONCERNED WITH THE REPRESENTATION CLAUSES. NONETHELESS, THE VISIBLE PART OF THE PACKAGE SPECIFICATION REFLECTS IDIOSYNCRASIES OF THE HARDWARE IN TWO WAYS:

- THE COUPLING OF LOGICALLY UNRELATED OPERATIONS IN SINGLE SUBPROGRAMS.

- AN UNNATURAL ORDERING FOR ENUMERATION LITERALS IN Motor_Setting_Type AND Tap_Extremity_Type. WERE WE NOT TIED TO A SPECIFIC ENCODING (WITH WHICH THE ABSTRACT ORDERING IS REQUIRED TO BE CONSISTENT), WE WOULD HAVE WRITTEN

  (At_Bottom, Between, At_Top)

  AND PROBABLY

  (Down, off, on)

  INSTEAD.

ANTENNA TUNER INTERFACE PACKAGE DECLARATION

```
package Antenna_Tuner_Package is

   type Motor_Setting_Type is (Off, Up, Down);
   type RF_Setting_Type is (RF_Disabled, RF_Enabled);
   type Band_Type is (Low_Band, High_Band);
   type Tap_Extremity_Type is (Between, At_Bottom, At_Top);
   type Tap_Position_Type is range 0 .. 255;

   procedure Control_Tuner
      (Motor_Setting : in Motor_Setting_Type;
       RF_Setting    : in RF_Setting_Type;
       Band          : in Band_Type);

   procedure Get_Tuner_Status
      (Tap_Extremity: out Tap_Extremity_Type; RF_Arriving: out Boolean);

   function Current_Tap_Position return Tap_Position_Type;

private

   for Motor_Setting_Type use (Off => 2#00#, Up => 2#01#, Down => 2#10#);
   for Motor_Setting_Type'Size use 2;

   for RF_Setting_Type use (RF_Disabled => 0, RF_Enabled => 1);
   for RF_Setting_Type'Size use 1;

   for Band_Type use (Low_Band => 0, High_Band => 1);
   for Band_Type'Size use 1;

   for Tap_Extremity_Type use
      (Between =  2#00#, At_Bottom => 2#01#, At_Top => 2#10#);
   for Tap_Extremity_Type'Size use 2;

   pragma Inline (Control_Tuner, Get_Tuner_Status, Current_Tap_Position);

end Antenna_Tuner_Package;
```

INSTRUCTOR NOTES

NAMED NUMBERS FOR CHANNELS WILL MAKE THE CALLS ON Low_Level_IO MORE READABLE. THEY ALSO MAKE IT EASY TO MODIFY THE PROGRAM IN THE EVENT CHANNELS ARE REASSIGNED.

CHANNEL NUMBERS ARE GIVEN IN HEXADECIMAL BECAUSE THAT IS HOW THEY ARE GIVEN IN THE SPECIFICATION. THIS MAKES IT EASY FOR A READER TO RELATE THE PROGRAM TO THE SPECIFICATION AND ALSO ELIMINATES THE POSSIBILITY OF AN ERROR IN TRANSLATING FROM HEXADECIMAL INTO DECIMAL.

THE SUBUNITS ARE GIVEN ON THE FOLLOWING SLIDES.

ANTENNA TUNER INTERFACE PACKAGE BODY

```
with Low_Level_IO, Unchecked_Conversion;

package body Antenna_Tuner_Package is

   Tuner_Control_Interface_Channel : constant := 16#10#;
   Tuner_Status_Interface_Channel  : constant := 16#10#;
   Tap_Position_Indicator_Channel  : constant := 16#20#;

   procedure Control_Tuner
      (Motor_Setting : in Motor_Setting_Type;
       RF_Setting    : in RF_Setting_Type;
       Band          : in Band_Type)
      is separate;

   procedure Get_Tuner_Status
      (Tap_Extremity: out Tap_Extremity_Type; RF_Arriving: out Boolean)
      is separate;

   function Current_Tap_Position return Tap_Position_Type is separate;

end Antenna_Tuner_Package;
```

INSTRUCTOR NOTES

THOUGH THIS IS A LOW-LEVEL ROUTINE, IT IS WRITTEN AS ABSTRACTLY AS POSSIBLE. THE TUNER CONTROL INTERFACE OUTPUT BYTE IS MODELED ABSTRACTLY AS A THREE-COMPONENT RECORD. THE REPRESENTATION CLAUSES TIE THE INTERNAL REPRESENTATION OF THE RECORD TO THE HARDWARE SPECIFICATION, BUT WITHIN THE SEQUENCE OF STATEMENTS Tuner_Control_Byte_Type IS USED AS AN ORDINARY RECORD TYPE.

(GO OVER THE EFFECT OF THE RECORD REPRESENTATION CLAUSE AND THE LENGTH CLAUSE.)

Control_Tuner COULD HAVE BEEN WRITTEN WITH A SINGLE PARAMETER OF Tuner_Control_Byte_Type RATHER THAN WITH THREE PARAMETERS BELONGING TO THE COMPONENT TYPES. THE FUNCTIONALITY OF THE PROCEDURE WOULD HAVE BEEN IDENTICAL. HOWEVER, IT WOULD HAVE BEEN LESS CONVENIENT FOR THE USER, WHO WOULD HAVE TO BUILD A RECORD BEFORE CALLING Control_Tuner. FURTHERMORE, THE INTERFACE OF Antenna_Tuner_Package WOULD HAVE BEEN COMPLICATED BY THE ADDITION OF ANOTHER TYPE KNOWN TO THE OUTSIDE WORLD.

THE TUNER CONTROL INTERFACE OUTPUT BYTE IS OBVIOUSLY NOT ONE OF THE VIEW OF A BYTE ANTICIPATED IN THE DESIGN OF Low_Level_IO. THUS Unchecked_Conversion IS NEEDED TO CONVERT THIS VIEW TO ONE OF THE VIEW THAT WAS ANTICIPATED. THE CHOICE OF Bit_Sequence_Byte AS THE TARGET TYPE IS ARBITRARY. Integer_Byte AND Character_Byte WOULD ALSO HAVE WORKED.

ANTENNA TUNER INTERFACE PACKAGE BODY -- Control_Tuner SUBUNIT

```
separate (Antenna_Tuner_Package)

procedure Control_Tuner
   (Motor_Setting : in Motor_Setting_Type;
    RF_Setting    : in RF_Setting_Type;
    Band          : in Band_Type) is

   type Tuner_Control_Byte_Type is
      record
         Motor_Setting_Part : Motor_Setting_Type;
         RF_Setting_Part    : RF_Setting_Type;
         Band_Part          : Band_Type;
      end record;
```

INSTRUCTOR NOTES

TAKES BITS OF Tuner_Control_Byte_Type AND INTERPRETS THEM AS TYPE Bit_Sequence SO YOU CAN USE Send_Control. See VG 22-2.

ANTENNA TUNER INTERFACE PACKAGE BODY -- Control_Tuner SUBUNIT (CONTINUED)

```
   for Tuner_Control_Byte_Type use
      record
         Motor_Setting_Part at 0 range 0 .. 1;
         RF_Setting_Part    at 0 range 2 .. 2;
         Band_Part          at 0 range 3 .. 3;
      end record;

   for Tuner_Control_Byte_Type'Size use 8;
   -- Converts Tuner_Control_Byte_Type to Bit_Sequence_Byte

   function Bit_Sequence_From_Tuner_Control_Byte is new
      Unchecked_Conversion
         (Source => Tuner_Control_Byte_Type,
          Target => Low_Level_IO.Bit_Sequence_Byte);

   Tuner_Control_Byte: Tuner_Control_Byte_Type;

begin -- Control_Tuner

   Tuner_Control_Byte :=
      (Motor_Setting_Part => Motor_Setting,
       RF_Setting_Part    => RF_Setting,
       Band_Part          => Band);
   -- Requires Bit_Sequence_Byte.  See VG 7-48.

   Low_Level_IO.Send_Control
      (Device => Tuner_Control_Interface_Channel,
       Data   => Bit_Sequence_From_Tuner_Control_Byte (Tuner_Control_Byte));

end Control_Tuner;
```

INSTRUCTOR NOTES

THIS SUBUNIT IS SIMILAR IN FORM AND SPIRIT TO THE PREVIOUS ONE.

ANTENNA TUNER INTERFACE PACKAGE BODY -- Get_Tuner_Status SUBUNIT

```
separate (Antenna_Tuner_Package)

procedure Get_Tuner_Status
   (Tap_Extremity: out Tap_Extremity_Type; RF_Arriving: out Boolean) is

   type Tuner_Status_Byte_Type is
      record
         RF_Arriving_Part   : Boolean;
         Tap_Extremity_Part : Tap_Extremity_Type;
      end record;

   for Tuner_Status_Byte_Type use
      record
         RF_Arriving_Part   at 0 range 0 .. 0;
         Tap_Extremity_Part at 0 range 1 .. 2;
      end record;
```

INSTRUCTOR NOTES

ANTENNA TUNER INTERFACE PACKAGE BODY -- Get_Tuner_Status SUBUNIT (CONTINUED)

```
   for Tuner_Status_Byte_Type'Size use 8;

   function Tuner_Status_From_Bit_Sequence_Byte is new
      Unchecked_Conversion
         (Source => Low_Level_IO.Bit_Sequence_Byte,
          Target => Tuner_Status_Byte_Type);

   Raw_Byte          : Low_Level_IO.Bit_Sequence_Byte;
   Tuner_Status_Byte : Tuner_Status_Byte_Type;

begin -- Get_Tuner_Status

   Low_Level_IO.Receive_Control
      (Device => Tuner_Status_Interface_Channel,
       Data   => Raw_Byte);

   Tuner_Status_Byte := Tuner_Status_From_Bit_Sequence_Byte (Raw_Byte);

   Tap_Extremity := Tuner_Status_Byte.Tap_Extremity_Part;
   RF_Arriving   := Tuner_Status_Byte.RF_Arriving_Part;

end Get_Tuner_Status;
```

INSTRUCTOR NOTES

THIS SUBUNIT IS SIMPLER THAN THE OTHERS FOR TWO REASONS:

- THE INTENDED ABSTRACT VIEW OF THE TAP POSITION INDICATOR OUTPUT BYTE -- AN INTEGER VALUE -- IS ONE ALREADY PROVIDED BY THE LANGUAGE. THERE IS NO NEED TO DEFINE A RECORD TYPE AND SPECIFY ITS REPRESENTATION.

- THIS VIEW OF A BYTE IS ANTICIPATED IN THE DESIGN OF Low_Level_IO, SO NO UNCHECKED CONVERSION IS NECESSARY.

THE RETURN STATEMENT CONTAINS AN ORDINARY ABSTRACT TYPE CONVERSION -- FROM THE SUBTYPE Low_Level_IO.Integer_Byte OF TYPE INTEGER TO THE INTEGER TYPE Tap_Position_Type DECLARED IN THE Antenna_Tuner_Package DECLARATION.

ANTENNA TUNER INTERFACE PACKAGE BODY -- Current_Tap_Position SUBUNIT

```
separate (Antenna_Tuner_Package)

function Current_Tap_Position return Tap_Position_Type is

   Raw_Byte: Low_Level_IO.Integer_Byte;

begin -- Current_Tap_Position

   Low_Level_IO.Receive_Control
      (Device => Tap_Position_Indicator_Channel,
       Data   => Raw_Byte);

   return Tap_Position_Type (Raw_Byte);

end Current_Tap_Position;
```

INSTRUCTOR NOTES

PART VII

REMAINING Ada FEATURES

INSTRUCTOR NOTES

THIS SECTION IS A BRIEF OVERVIEW OF TASKING GIVEN TO EXPOSE STUDENTS TO THE REMAINING FEATURES OF ADA. MORE COMPLETE COVERAGE IS GIVEN IN MODULES L303 AND L401.

THIS SECTION CAN BE SKIPPED WHEN L305 IS BEING TAKEN AS A PREREQUISITE FOR L303 AND L401.

SECTION 23

OVERVIEW OF Ada TASKING

INSTRUCTOR NOTES

BULLET 1: POINT OUT THE DISTINCTION BETWEEN A SEQUENCE OF INSTRUCTIONS, OR PROGRAM (STATIC) AND A SEQUENCE OF ACTIONS, OR TASK (DYNAMIC). INSTRUCTIONS ARE TEXT AND ACTIONS ARE EVENTS.

BULLET 2: AN OPERATING SYSTEM OR RUNTIME SUPPORT ENVIRONMENT IS RESPONSIBLE FOR ACHIEVING INTERLEAVING.

THE RULES OF Ada DO NOT SPECIFY WHETHER TASKS ARE EXECUTED SIMULTANEOUSLY OR INTERLEAVED. THAT DEPENDS ON THE IMPLEMENTATION.

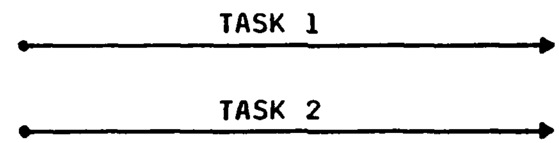
(ANOTHER POSSIBILITY IS A COMBINATION OF THE TWO METHODS.)

BULLET 3: IN GENERAL, NOTHING MAY BE ASSUMED IN ONE TASK ABOUT THE PROGRESS THAT HAS BEEN MADE BY ANOTHER TASK.

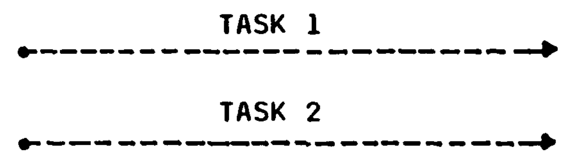
WHEN Ada TASKS COMMUNICATE, THE FIRST ONE TO REACH A POINT WHERE IT EXPECTS COMMUNICATION WAITS FOR THE OTHER TASK TO REACH A SIMILAR POINT. A LATER SLIDE DESCRIBES THIS IN MORE DETAIL.

WHAT IS A TASK?

- A TASK IS A <u>SEQUENCE OF ACTIONS</u> PERFORMED IN CARRYING OUT A PROGRAM.
- SEVERAL TASKS CAN BE IN PROGRESS AT THE SAME TIME.
  - -- SIMULTANEOUS EXECUTION OF DIFFERENT PROCESSORS



  - -- INTERLEAVED EXECUTION ON A SINGLE PROCESSOR, GIVING THE <u>APPEARANCE</u> OF SIMULTANEITY



- TASKS ARE <u>ASYNCHRONOUS</u>.
  - -- EACH PROCEEDS AT A SPEED INDEPENDENT OF THE OTHERS.
  - -- THIS CAN CREATE PROBLEMS WHEN ONE TASK TRIES TO USE DATA BEING PRODUCED BY ANOTHER TASK.
  - -- Ada TASKS SYNCHRONIZE MOMENTARILY WHEN THEY COMMUNICATE
- UNTIL NOW YOU HAVE ONLY SEEN PROGRAMS MEANT TO BE EXECUTED BY A SINGLE TASK.

INSTRUCTOR NOTES

THE FIRST SUB-BULLET OF BULLET 2 APPLIES TO AN OPERATING SYSTEM, THE SECOND TO AN EMBEDDED SYSTEM.

IN THE THIRD BULLET, THE CIRCLES REPRESENT TASKS AND THE ARROWS REPRESENT STREAMS OF INPUT AND OUTPUT DATA.

-- TASK 1 READS A SEQUENCE OF 65-CHARACTER LINES AND OUTPUTS THE CONTENTS OF THE LINES CHARACTER-BY-CHARACTER, OUTPUTTING AN EXTRA BLANK AT THE END OF EACH CARD (BECAUSE THE END OF A CARD SHOULD SERVE AS A "LOGICAL BLANK," SEPARATING WORDS.)

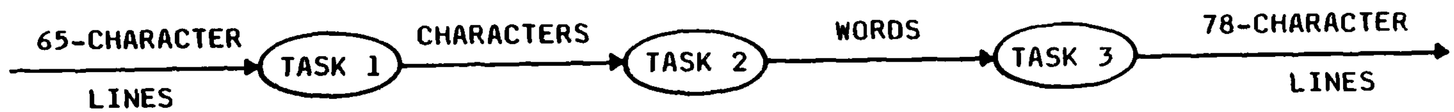
-- TASK 2 INPUTS A STREAM OF BLANK AND NON-BLANK CHARACTERS, OBLIVIOUS TO THEIR ORIGINAL SOURCE, GROUPS CONSECUTIVE NON-BLANK CHARACTERS INTO WORDS, AND OUTPUTS A STREAM OF WORDS.

-- TASK 3 INPUTS A STREAM OF WORDS, OBLIVIOUS TO THEIR ORIGINAL SOURCE OR HOW THEY WERE SEPARATED, GROUPS THEM INTO 78-CHARACTER LINES, AND WRITES A SEQUENCE OF LINES.

THE DISTINCTION BETWEEN THE SECOND AND THIRD REASONS FOR MULTITASK PROGRAMS IS NOT ALWAYS SHARP.

SOME REASONS FOR MULTITASK PROGRAMS

- SAVE TIME BY LETTING DIFFERENT PROCESSORS WORK ON DIFFERENT PARTS OF THE SYSTEM SIMULTANEOUSLY.
  - -- SORT TWO HALVES OF AN ARRAY SIMULTANEOUSLY, THEN MERGE THEM.
  - -- TIME IS ONLY REALLY SAVED WHEN TASKS ARE RUNNING ON SEPARATE PROCESSORS.
- MANAGE SEVERAL REAL-WORLD ACTIVITIES GOING ON SIMULTANEOUSLY.
  - -- TERMINALS, DISK DRIVES, AND TAPE DRIVES
  - -- SENSORS, DISPLAYS, AND MOTORS
- LOGICALLY DECOMPOSE A PROBLEM INTO SEVERAL INDEPENDENT "THREADS OF CONTROL," EACH WITH A SINGLE STREAM OF INPUTS AND OUTPUTS.
  - -- REFORMAT 65-COLUMN LINES OF TEXT TO 78 COLUMNS:



  - -- EACH INDIVIDUAL THREAD OF CONTROL IS MUCH EASIER TO PROGRAM BECAUSE IT SOLVES A MORE SIMPLY-STRUCTURED PROBLEM.

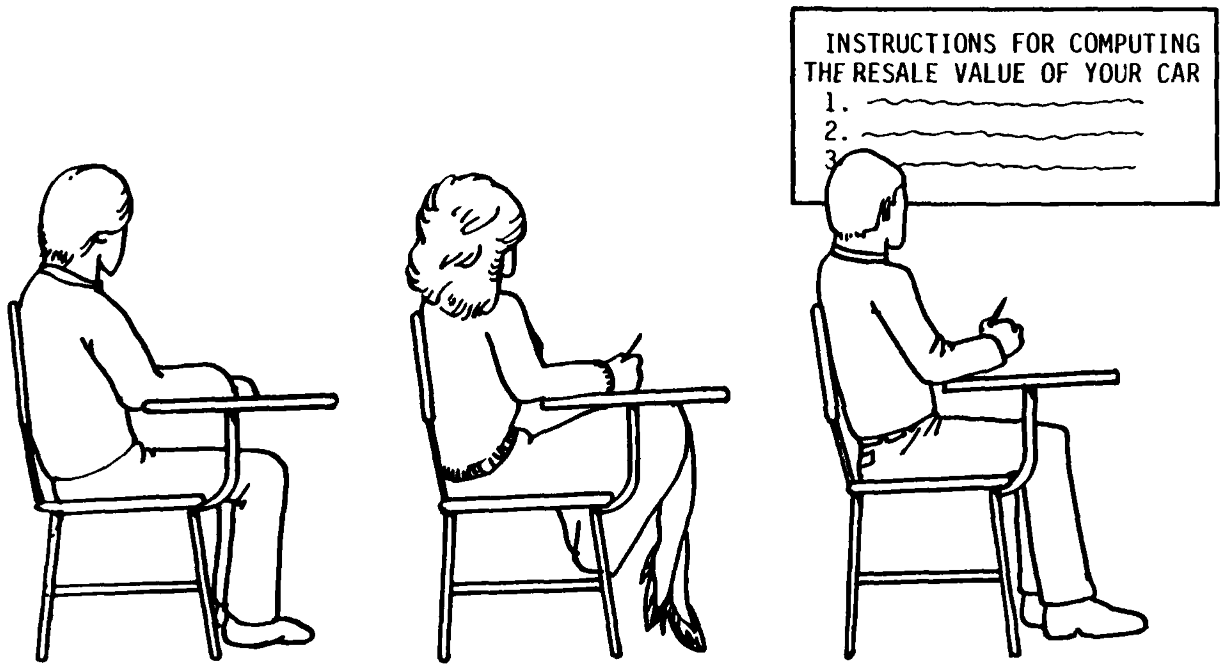
INSTRUCTOR NOTES

THE INSTRUCTIONS ON THE BLACKBOARD REPRESENT A PROGRAM. THE STUDENTS SITTING AT THEIR DESKS REPRESENT TASKS (OR, MORE PRECISELY, PROCESSORS EXECUTING TASKS).

ALL STUDENTS CAN SHARE THE SAME INSTRUCTIONS BECAUSE THEY DON'T MODIFY THE PROGRAM. EACH STUDENT USES HIS OWN SCRAP PAPER RATHER THAN THE SHARED BLACKBOARD TO PERFORM COMPUTATIONS.

PROGRAMS WRITTEN IN SUCH A WAY THAT THEY DO NOT MODIFY THEMSELVES BUT CAN BE SHARED BY SEVERAL TASKS, EACH USING THEIR OWN DATA AREAS, ARE CALLED RE-ENTRANT. Ada COMPILERS ARE REQUIRED TO GENERATE ONLY RE-ENTRANT OBJECT CODE.

SEVERAL TASKS CAN EXECUTE THE SAME INSTRUCTIONS



- EACH TASK MUST HAVE A SET OF DATA RESERVED FOR ITS EXCLUSIVE USE, WITH ITS OWN COPY OF THE VARIABLES CHANGED BY THE PROGRAM.

- THE SHARED COPY OF THE INSTRUCTIONS MUST NOT BE CHANGED BY ANY OF THE TASKS.

INSTRUCTOR NOTES

THE IDEA OF A TASK AS A DATA OBJECT MAY BE QUITE DIFFICULT FOR STUDENTS TO DIGEST. THEY ARE PROBABLY USED TO THINKING OF DATA AS SOMETHING THAT JUST SITS PASSIVELY UNTIL MANIPULATED BY AN OPERATION. ASK THEM TO TRY GENERALIZING FROM A NATURAL HISTORY MUSEUM, IN WHICH THE BOXES CONTAIN STUFFED ANIMALS, TO A ZOO, IN WHICH THE BOXES CONTAIN LIVE, ACTIVE ANIMALS. IF THIS FAILS (AS WELL IT MIGHT), ASK THEM TO ACCEPT THE FORMALISM FOR NOW, EVEN IF IT DOESN'T MAKE SENSE INTUITIVELY. EXAMPLES ON SUBSEQUENT SLIDES WILL, IT IS HOPED, MAKE THE NOTION OF TASK OBJECTS AS DATA SEEM MORE NATURAL.

THE "IMPLICIT AGENT" LISTED AS THE FOURTH "COMPONENT" OF THE TASK OBJECT IS A <u>VIRTUAL PROCESSOR</u> DEDICATED TO EXECUTING THE TASK. SEVERAL VIRTUAL PROCESSORS MAY BE IMPLEMENTED WITH TIME SLICES ON A SINGLE PHYSICAL PROCESSOR.

IN Ada, TASKS ARE VIEWED AS DATA OBJECTS

- A TASK OBJECT CONSISTS OF:
  - -- A SET OF INSTRUCTIONS
  - -- A SET OF VARIABLES AND OTHER DATA FOR USE ONLY BY THIS TASK
  - -- AN INTERFACE FOR COMMUNICATION WITH OTHER TASKS
  - -- AN IMPLICIT AGENT TO EXECUTE THE INSTRUCTIONS USING THE VARIABLES

- UNLIKE OTHER DATA OBJECTS, TASK OBJECTS ARE ACTIVE, CHANGING ENTITIES.

- THE TERMS <u>TASK</u> AND TASK OBJECT ARE USED INTERCHANGEABLY.

INSTRUCTOR NOTES

BULLET 1: IF ITEM 2 SEEMS CONFUSING, DRAW A PARALLEL TO RECORD TYPES. EACH OBJECT IN A RECORD TYPE IS DESCRIBED BY THE SAME COMPONENT DECLARATIONS, BUT EACH OBJECT HAS A SEPARATE COPY OF THE COMPONENTS.

EACH OBJECT IN A TASK TYPE HAS ITS OWN VIRTUAL PROCESSOR.

BULLET 3: LIKE ANY TYPE, A TASK TYPE CAN BE DESCRIBED IN TERMS OF ITS VALUES AND OPERATIONS. THE VALUES WERE DESCRIBED ON THE PREVIOUS SLIDE. THE OPERATIONS INCLUDE PRIMARILY ENTRY CALLS ON A TASK, BUT ALSO TASK ACTIVATION, TASK ABORTION, AND ATTRIBUTES.

BULLET 4: THE TERMINAL-HANDLER TASK OBJECTS MAY BE COMPONENTS OF AN ARRAY OF TASK OBJECTS.

TASK OBJECTS BELONG TO TASK TYPES

- THE OBJECTS IN A TASK TYPE SHARE THE FOLLOWING CHARACTERISTICS:
  - -- THE SAME SET OF INSTRUCTIONS
  - -- THE SAME SET OF DECLARATIONS (BUT EACH TASK OBJECT IN THE TASK TYPE HAS ITS OWN COPY OF THE VARIABLES DESCRIBED BY THESE DECLARATIONS)
  - -- THE SAME INTERFACE FOR COMMUNICATING WITH OTHER TASKS
- TASK TYPES ARE LIMITED TYPES
  - -- TASK OBJECTS MAY NOT BE ASSIGNED OR COMPARED FOR EQUALITY AND INEQUALITY
  - -- TASK TYPES MAY BE USED AS COMPONENT TYPES FOR ARRAYS AND RECORDS, BUT THEN THE RESULTING ARRAY TYPE OR RECORD TYPE IS ALSO LIMITED
  - -- TASK TYPES MAY BE USED IN ACCESS TYPE DECLARATIONS. THE RESULTING ACCESS TYPE IS <u>NOT</u> LIMITED. (POINTERS TO TASK OBJECTS MAY BE ASSIGNED AND COMPARED FOR EQUALITY AND INEQUALITY.)
- PRINCIPAL OPERATIONS ON OBJECTS OF A TASK TYPE ARE COMMUNICATIONS WITH THE EXECUTING TASK OBJECT.
- EXAMPLE:
  - -- A TASK TYPE FOR TASKS HANDLING TERMINALS
  - -- ONE TASK OBJECT FOR EACH TERMINAL
  - -- THEN ALL TERMINALS HAVE THEIR OWN TERMINAL-HANDLER TASKS, ALL EXECUTING THE SAME PROGRAM OF THEIR OWN PACE WITH THEIR OWN DATA
  - -- OPERATIONS MAY INCLUDE SENDING AND RECEIVING LINES OF DATA

INSTRUCTOR NOTES

DECLARING TASK TYPES

- TWO STEPS - TASK TYPE DECLARATION AND TASK BODY.
- TASK TYPE DECLARATION DESCRIBES THE INTERFACE FOR COMMUNICATION WITH OTHER TASKS.
- TASK BODY CONTAINS THE DATA DECLARATIONS AND INSTRUCTIONS FOR TASKS IN THE TASK TYPE.
- THE TASK TYPE DECLARATION AND TASK BODY TOGETHER ARE CALLED A <u>TASK UNIT</u>.

INSTRUCTOR NOTES

BULLET 1: THIS BASIC FORM FOR TASK TYPE DECLARATIONS DOES NOT ALLOW FOR TASK OBJECTS IN ANONYMOUS TASK TYPES, ENTRY FAMILIES, OR REPRESENTATION CLAUSES. THESE ARE DESCRIBED LATER.

BULLET 4: ENTRIES ARE DISCUSSED IN GREATER DETAIL LATER.

BULLET 6: IF PRESSED BY STUDENT WHO INSIST THAT A TASK WITHOUT ENTRY DECLARATIONS IS USELESS, REPLY AS FOLLOWS: ENTRY DECLARATIONS ONLY SPECIFY HOW OTHER TASK OBJECTS INITIATE COMMUNICATION WITH TASK OBJECTS IN THIS TYPE; A TASK WITHOUT ENTRIES MAY STILL INITIATE COMMUNICATION WITH OTHER TASKS BASED ON THE ENTRY DECLARATIONS FOR THE OTHER TASK'S TYPE. DEFER THIS ISSUE IF NOT PRESSED TO COVER IT.

TASK TYPE DECLARATIONS

- BASIC FORM:

```
task type [identifier] is
   {entry [identifier] [([formal parameters])];}
end [[identifier]];
```

- THE FIRST IDENTIFIER IS THE NAME OF THE TASK TYPE.
- THE LAST IDENTIFIER MUST MATCH THE FIRST.
  - -- AS A MATTER GOOD PROGRAMMING STYLE, IT SHOULD NEVER BE OMITTED.
- THE FIRST AND LAST LINES SURROUND ZERO OR MORE <u>ENTRY DECLARATIONS</u>.
  - -- AN ENTRY IS THE MEANS BY WHICH OTHER TASKS COMMUNICATE WITH A TASK IN THIS TYPE.
  - -- ENTRY DECLARATIONS LOOK LIKE PROCEDURE DECLARATIONS, BUT WITH THE WORD <u>PROCEDURE</u> REPLACED BY <u>ENTRY</u>.
- EXAMPLE:

```
task type Message_Buffer_Type is
   entry Send_Message (Message: in Message_Type);
   entry Receive_Message (Message: out Message_Type);
end Message_Buffer_Type;
```

- SHORTHAND TASK-TYPE DECLARATION FOR TASK TYPES WITH NO ENTRIES.

```
task type [identifier] ;
```

INSTRUCTOR NOTES

TASK BODIES DIFFER FROM SUBPROGRAM BODIES, PACKAGE BODIES, AND BLOCK STATEMENTS ONLY IN THE FORM OF THE "HEADER LINE" AND IN WHICH PARTS ARE OPTIONAL.

THE STATEMENTS ALLOWED ONLY IN TASK BODIES ARE DESCRIBED ON LATER SLIDES.

TASK BODIES

- SAME BASIC FORM AS SUBPROGRAM BODIES, PACKAGE BODIES, AND BLOCK STATEMENTS:

```
    task body [identifier] is
       [declarative part]
    begin
       [sequence of statements]
   [exception
       [sequence of exception handlers] ]
    end [[identifier]];
```

- THE IDENTIFIERS OF THE TOP AND BOTTOM MUST BE THE NAME OF THE TASK TYPE.
  - -- AGAIN, AS A MATTER OF STYLE, THE BOTTOM IDENTIFIER SHOULD NOT BE OMITTED.

- THE DECLARATIVE PART ACTS AS A TEMPLATE. EACH OBJECT IN THE TASK TYPE WILL BE GIVEN A COPY OF THE DATA DESCRIBED THERE.

- A TASK BODY MAY CONTAIN CERTAIN STATEMENTS NOT ALLOWED ELSEWHERE IN AN Ada PROGRAM.

INSTRUCTOR NOTES

BULLET 1: THE RESTRICTION ON WHAT MAY APPEAR AFTER THE TASK BODY IS A RULE THAT APPLIES TO SUBPROGRAM BODIES, PACKAGE BODIES, AND BODY STUBS AS WELL.

BULLET 2: WHEN THE TASK TYPE DECLARATION AND TASK BODY BOTH GO IN THE DECLARATIVE PART OF A PACKAGE BODY, THE TASK TYPE IS ONLY FOR INTERNAL USE IN IMPLEMENTING THE PACKAGE. WHEN THE TASK TYPE DECLARATION GOES IN THE VISIBLE PART OF A PACKAGE DECLARATION, IT IS PROVIDED BY THE PACKAGE TO ALL USERS OF THE PACKAGE.

A LIMITED PRIVATE TYPE DECLARATION IN THE VISIBLE PART OF A PACKAGE CAN BE ACCOMPANIED BY A TASK TYPE DECLARATION FOR THE SAME TYPE IN THE PRIVATE PART AND BY A TASK BODY IN THE PACKAGE BODY. (THE TASK'S ENTRIES CAN ONLY BE CALLED FROM WITHIN THE PACKAGE BODY, BUT SUBPROGRAMS OPERATING ON TASK TYPE OBJECTS CAN BE IMPLEMENTED IN TERMS OF ENTRY CALLS.)

BULLET 5: A LIBRARY UNIT IS A SEPARATELY COMPILED UNIT MADE AVAILABLE WITH A <u>with</u> CLAUSE.

WHERE TASK UNITS GO

- IN THE DECLARATIVE PART OF A BLOCK STATEMENT, SUBPROGRAM BODY, PACKAGE BODY, OR OUTER TASK BODY
  - -- THE TASK TYPE DECLARATION APPEARS FIRST
  - -- THE TASK BODY APPEARS SOME PLACE BELOW THE DECLARATION, BUT NONE OF THE FOLLOWING MAY APPEAR AFTER THE BODY:
    - USE CLAUSES
    - DECLARATIONS OF OBJECTS, NAMED NUMBERS, TYPES, SUBTYPES, AND EXCEPTIONS
    - RENAMING DECLARATIONS
- PROVIDED BY A PACKAGE
  - -- TASK TYPE DECLARATION IN THE PACKAGE DECLARATION
  - -- TASK BODY IN THE PACKAGE BODY
- TASK <u>OBJECT</u> DECLARATIONS ARE POSSIBLE IMMEDIATELY FOLLOWING THE TASK TYPE DECLARATION.
- TASK BODIES CAN BE REPLACED BY BODY STUBS AND COMPILED SEPARATELY AS SUBUNITS.
  - -- `task body` $\boxed{\texttt{identifier}}$ `is separate;`
- TASK TYPE DECLARATIONS AND TASK BODIES MAY <u>NOT</u> BE COMPILED SEPARATELY AS LIBRARY UNITS.
  - -- ONE CAN ACHIEVE ALMOST THE SAME EFFECT BY WRITING A LIBRARY PACKAGE PROVIDING ONLY A TASK UNIT.

INSTRUCTOR NOTES

THIS SLIDE IS MEANT TO HELP THE STUDENT INTEGRATE TASK UNITS INTO HIS UNDERSTANDING OF Ada. IT SHOWS THAT TASK UNITS FIT INTO THE LANGUAGE IN A MANNER CONSISTENT WITH THE FEATURES THAT WERE TAUGHT EARLIER.

BULLET 2:

SUB-BULLET 1: IN CERTAIN CONTEXTS, SUBPROGRAM DECLARATIONS MAY BE OMITTED, AND SUBPROGRAM BODIES SERVE BOTH ROLES.

PROGRAM UNITS

- Ada PROVIDES FOUR KINDS OF PROGRAM UNITS:

  -- SUBPROGRAMS
  -- PACKAGES
  -- GENERIC UNITS
  -- TASK UNITS

- SIMILARITIES AMONG PROGRAM UNITS:

  -- EACH HAS TWO PARTS: A DECLARATION DESCRIBING THE EXTERNAL VIEW AND A BODY DESCRIBING THE IMPLEMENTATION.

	SUBPROGRAMS	PACKAGES	GENERIC UNITS	TASK UNITS
EXTERNAL VIEW	SUBPROGRAM DECLARATION	PACKAGE DECLARATION	GENERIC SUBPROGRAM OR GENERIC PACKAGE DECLARATION	TASK TYPE DECLARATION
IMPLEMENTATION	SUBPROGRAM BODY	PACKAGE BODY	SUBPROGRAM OR PACKAGE BODY	TASK BODY

INSTRUCTOR NOTES

BULLET 1: A CALL ON A SUBPROGRAM MAY APPEAR AFTER THE SUBPROGRAM DECLARATION AND BEFORE THE BODY; AN ENTITY PROVIDED BY A PACKAGE MAY BE REFERRED TO AFTER THE PACKAGE DECLARATION AND BEFORE THE PACKAGE BODY; A GENERIC UNIT MAY BE INSTANTIATED AFTER THE GENERIC DECLARATION AND BEFORE THE GENERIC BODY; OBJECTS IN A TASK TYPE MAY BE DECLARED AFTER THE TASK TYPE DECLARATION AND BEFORE THE TASK BODY.

PROGRAM UNITS (Continued)

- SIMILARITIES (CONTINUED)
  - EXTERNAL VIEW AND BODY MAY BOTH APPEAR IN A DECLARATIVE PART.
  - EXTERNAL VIEW CAN BE GIVEN IN A PACKAGE DECLARATION WITH INTERNAL VIEW IN A PACKAGE BODY.
  - EXTERNAL VIEW IS SUFFICIENT TO ALLOW USE OF THE UNIT.
- DIFFERENCES FOR TASK UNITS:
  - FOR SUBPROGRAMS, GENERIC UNITS, AND PACKAGES, EXTERNAL VIEW MAY BE COMPILED SEPARATELY AS A LIBRARY UNIT (TO BE MADE AVAILABLE THROUGH A <u>with</u> CLAUSE) WITH THE INTERNAL VIEW COMPILED LATER AS A SECONDARY UNIT.
  - THERE ARE NO GENERIC TASK UNITS. (ONE CAN WRITE A GENERIC PACKAGE PROVIDING ONLY A TASK TYPE DECLARATION.)

INSTRUCTOR NOTES

THIS EXAMPLE MAY BE HELPFUL FOR STUDENTS WHO HAVE TROUBLE THINKING OF TASKS AS DATA. A MESSAGE BUFFER IS AN OBJECT WITH TWO OPERATIONS: SENDING A MESSAGE TO A BUFFER, AND RECEIVING THE OLDEST YET-UNRECEIVED MESSAGE SENT TO THE BUFFER.

THE TASK BODY WILL BE GIVEN LATER.

(ASSUME THE TYPE Message_Type IS DECLARED GLOBALLY SOMEWHERE.)

EXAMPLE OF TASK OBJECT DECLARATIONS

- THE FOLLOWING PACKAGE PROVIDES A TASK TYPE AND TWO OBJECTS BELONGING TO THE TYPE.

```
package Message_Buffer_Package is

   task type Message_Buffer_Type is
      entry Send (Message : in Message_Type);
      entry Receive (Message : out Message_Type);
   end Message_Buffer_Type;

   Buffer_1, Buffer_2 : Message_Buffer_Type;

end Message_Buffer_Package;

package body Message_Buffer_Package is

   task body Message_Buffer_Type is
      ...
   begin -- Message_Buffer_Type
      ...
   end Message_Buffer_Type;

end Message_Buffer_Package;
```

INSTRUCTOR NOTES

THE PURPOSE OF THIS REVIEW IS TO PREPARE STUDENTS FOR THE ANALOGY TO ANONYMOUS TASK TYPES ON THE NEXT SLIDE.

REVIEW -- ANONYMOUS ARRAY TYPES

- Ada PROVIDES A SHORTHAND FOR DECLARING "ONE-OF-A-KIND" ARRAYS.

- THE DECLARATIONS

```
type Days_In_Month_Type is array (Month_Type) of Positive;
Days_In_Month : Days_In_Month_Type;
```

  CAN BE ABBREVIATED BY A SPECIAL KIND OF OBJECT DECLARATION:

```
Days_In_Month : array (Month_Type) of Positive;
```

- THE OBJECT (Days_In_Month) IS SAID TO BELONG TO AN <u>ANONYMOUS ARRAY TYPE</u>.

INSTRUCTOR NOTES

POINT OUT THE WORD <u>type</u> IN THE DECLARATION OF Message_Buffer_Type. MAKE SURE THE CLASS UNDERSTANDS THAT THE ONLY SYNTACTIC DIFFERENCE IN THE SECOND VERSION IS THE OMISSION OF THIS WORD.

LAST BULLET: THIS IS IDENTICAL TO THE DECLARATION OF A TASK <u>type</u> WITH NO ENTRIES, EXCEPT THAT THE WORD <u>type</u> IS AGAIN OMITTED.

OF COURSE THIS MUST BE FOLLOWED BY A TASK BODY.

ANONYMOUS TASK TYPES

- THERE IS AN ANALOGOUS SHORTHAND FOR ONE-OF-A-KIND TASK OBJECTS.
- THE DECLARATIONS

```
task type Message_Buffer_Type is
   ...
end Message_Buffer_Type;

Message_Buffer : Message_Buffer_Type;

task body Message_Buffer_Type is
   ...
end Message_Buffer_Type
```

CAN BE ABBREVIATED AS FOLLOWS:

```
task Message_Buffer is
   ...
end Message_Buffer;

task body Message_Buffer is
   ...
end Message_Buffer;
```

- BECAUSE THE WORD <u>type</u> IS MISSING FOLLOWING THE WORD <u>task</u> IN THE DECLARATION, THIS DECLARES A SINGLE TASK OBJECT RATHER THAN A TASK TYPE.
- THE TASK OBJECT (Message_Buffer) IS SAID TO BELONG TO AN <u>ANONYMOUS TASK TYPE</u>.
- DECLARATION FOR ONE-OF-A-KIND TASK OBJECTS WITH NO ENTRIES:

```
task identifier ;
```

INSTRUCTOR NOTES

THE FOLLOWING SLIDE ILLUSTRATES THE RULES GIVEN IN THIS SLIDE.

RULE IS BASED ON WHERE TASK <u>OBJECTS</u> ARE DECLARED.

ACTIVATION AND TERMINATION OF TASKS

- ACTIVATION OF A TASK BEGINS THE SEQUENCE OF ACTIONS SPECIFIED BY THE TASK BODY:

  -- FIRST THE DECLARATIONS IN THE TASK BODY ARE ELABORATED

  -- THEN THE STATEMENTS IN THE TASK BODY ARE EXECUTED

- FOR A TASK OBJECT DECLARED IN THE DECLARATIVE PART OF A BLOCK STATEMENT, SUBPROGRAM BODY, OR TASK BODY:

  -- ACTIVATION OCCURS AFTER ELABORATION OF THAT DECLARATIVE PART, BEFORE THE CORRESPONDING SEQUENCE OF STATEMENTS BEGINS EXECUTION.

  -- DEPARTURE FROM THE CORRESPONDING SEQUENCE OF STATEMENTS AWAITS COMPLETION OF THE STATEMENTS PLUS TERMINATION OF THE TASK.

INSTRUCTOR NOTES

THE FOLLOWING SLIDE ILLUSTRATES THE RULES GIVEN IN THIS SLIDE.

BULLET 1'S RULE IS BASED ON WHERE ACCESS <u>TYPES</u> DESIGNATING TASK OBJECTS ARE DECLARED.

BULLET 2'S RULE MAY HAVE TO BE APPLIED SEVERAL TIMES IN SUCCESSION IN THE CASE OF PACKAGES NESTED IN PACKAGES NESTED IN PACKAGES ...

BULLET 3: SIMILARLY, THE SAME RULES APPLY AS IF AN ACCESS TYPE DESIGNATING A SINGLE TASK OBJECT WERE DECLARED AT A POINT WHERE AN ACCESS TYPE DESIGNATING THE COMPOSITE OBJECT IS DECLARED.

ACTIVATION AND TERMINATION OF TASKS

- FOR A DYNAMICALLY ALLOCATED TASK OBJECT:

  -- ACTIVATION OCCURS UPON ELABORATION OF THE ALLOCATOR

  -- DEPARTURE FROM THE BLOCK STATEMENT, SUBPROGRAM BODY, OR TASK BODY DECLARING THE CORRESPONDING <u>ACCESS TYPE</u> AWAITS COMPLETION OF ITS STATEMENTS PLUS TERMINATION OF ALL TASKS DESIGNATED BY VALUES IN THAT ACCESS TYPE.

- FOR TASK OBJECTS OR ACCESS TYPES DECLARED IN PACKAGES:

  -- FOR LIBRARY PACKAGES, ACTIVATION OCCURS BEFORE EXECUTION OF THE MAIN PROGRAM, AND NOTHING WAITS FOR COMPLETION OF THE TASK.

  -- FOR PACKAGES IN DECLARATIVE PARTS, THE RULES ARE AS IF THE DECLARATIONS INSIDE THE PACKAGE OCCURRED DIRECTLY IN THE DECLARATIVE PART.

- FOR A TASK OBJECT THAT IS THE COMPONENT OF SOME ARRAY OR RECORD:

  -- THE SAME RULES APPLY AS IF A SEPARATE TASK OBJECT WERE DECLARED OR ALLOCATED AT THE POINT WHERE THE COMPOSITE OBJECT IS DECLARED OR ALLOCATED.

INSTRUCTOR NOTES

THIS IS AN ARTIFICIAL EXAMPLE CONTRIVED TO ILLUSTRATE THE RULES.

RUN THROUGH THE DECLARATIONS IN THE DECLARATIVE PART OF EXAMPLE.

TWELVE TASK OBJECTS ARE ACTIVATED JUST AS THE SEQUENCE OF STATEMENTS IN EXAMPLE IS ENTERED : Task_1 BECAUSE IT IS DECLARED IN A PACKAGE AND THEREFORE TREATED AS IF THE OBJECT DECLARATION OCCURRED WHERE THE PACKAGE DECLARATION OCCURS; THE TEN TASK OBJECT COMPONENTS OF Task_List, BECAUSE Task_List IS A COMPOSITE OBJECT DECLARED IN THE DECLARATIVE PART OF EXAMPLE. (HAD Task_List BEEN A RECORD OBJECT WITH A Sample_Task Type COMPONENT, THE SAME RULE WOULD HAVE APPLIED.)

RUN THROUGH THE DECLARATIONS IN THE DECLARATIVE PART OF THE BLOCK STATEMENT.

Task_3 IS ACTIVATED JUST AS THE SEQUENCE OF STATEMENTS IN THE BLOCK STATEMENT IS ENTERED, BECAUSE IT IS A TASK OBJECT DECLARED IN THE CORRESPONDING DECLARATIVE PART.

THOUGH THE TWO ALLOCATORS IN THE BLOCK STATEMENT APPEAR IDENTICAL, THEY HAVE DIFFERENT TYPES. THE FIRST CREATES A First_Task_Pointer_Type VALUE DESIGNATING A Sample_Task_Type OBJECT AND THE SECOND CREATES A Second_Task_Pointer_Type VALUE DESIGNATING A Sample_Task_Type OBJECT. BOTH ALLOCATED TASK OBJECTS ARE ACTIVATED AS THEY ARE ALLOCATED.

ILLUSTRATION OF TASK ACTIVATION AND TERMINATION

```
procedure Example is

   task type Sample_Task_Type;
   type First_Task_Pointer_Type is access Sample_Task_Type;
   Task_1 : Sample_Task_Type;
   package Task_2_Package is
      Task_2 : Sample_Task_Type;
   end Task_2_Package;
   Task_List : array (1 .. 10) of Sample_Task_Type;
   task body Sample_Task_Type is separate;

begin -- Example            ACTIVATION OF Task_1, Task_2_Package.Task_2,
                            AND Task_List (1) THROUGH Task_List (10)
   ...
   declare

      type Second_Task_Pointer_Type is access Sample_Task_Type;
      Task_Pointer_1 : First_Task_Pointer_Type;
      Task_Pointer_2 : Second_Task_Pointer_Type;
      Task_3         : Sample_Task_Type;
   begin -- Block statement   ACTIVATION OF Task_3
                              ACTIVATION OF ALLOCATED TASK
      Task_Pointer_1 :=  new Sample_Task_Type;
                              ACTIVATION OF ALLOCATED TASK
      Task_Pointer_2 :=  new Sample_Task_Type;
         ...
```

DEPARTURE FROM THE BLOCK STATEMENT AWAITS TERMINATION OF Task_3 BECAUSE Task_3 IS DECLARED IN THE BLOCK STATEMENT'S DECLARATIVE PART; AND TERMINATION OF Task_Pointer_2.all BECAUSE THE ACCESS VALUE DESIGNATING THAT TASK OBJECT BELONGS TO AN ACCESS TYPE DECLARED IN THE BLOCK STATEMENT'S DECLARATIVE PART. TERMINATION OF Task_Pointer_1.all IS NOT AWAITED (EVEN THOUGH IT WAS ACTIVATED IN THE BLOCK STATEMENT) BECAUSE THE ACCESS TYPE DESIGNATING IT WAS NOT DECLARED IN THE BLOCK STATEMENT.

DEPARTURE FROM EXAMPLE AWAITS TERMINATION OF Task_1 BECAUSE IT IS DECLARED IN THE DECLARATIVE PART OF EXAMPLE; OF Task_2_Package.Task_2 AND Task_List (1) THROUGH Task_List (10) BECAUSE THEY ARE TREATED AS IF THEY WERE DECLARED THERE; OF Task_Pointer_1.all BECAUSE IT IS DESIGNATED BY A VALUE IN AN ACCESS TYPE DECLARED THERE.

IF EXECUTION OF EXAMPLE IS ENDED BY A RETURN STATEMENT, COMPLETION OF THE EXECUTION OF THIS RETURN STATEMENT AWAITS TERMINATION OF THOSE TASKS.

HAD First_Task_Pointer_Type OR Second_Task_Pointer_Type INSTEAD POINTED TO A COMPOSITE TYPE WITH A TASK TYPE COMPONENT, THE RESULTS WOULD HAVE BEEN THE SAME.

ILLUSTRATION OF TASK ACTIVATION AND TERMINATION (Continued)

```
         -- AWAIT TERMINATION OF Task_3, Task_Pointer_2.all

      end; -- Block statement
      ...

      -- AWAIT TERMINATION OF Task_1, Task_2_Package.Task_2, Task_List (1) THROUGH
      -- Task_List (10), AND Task_1_Pointer.all

   end Example;
```

INSTRUCTOR NOTES

BULLET 2: THIS "USUAL FORM" DOES NOT ACCOMMODATE ENTRY FAMILIES OR ENTRIES RENAMED AS PROCEDURES.

BULLET 4: ACCEPT STATEMENTS ARE COVERED ON THE NEXT SLIDE. ENTRY PARAMETERS OF MODE <u>in</u> OR <u>in out</u> PASS INFORMATION FROM THE CALLING TASK TO THE CALLED TASK. ENTRY PARAMETERS OF MODE <u>in out</u> OR <u>out</u> PASS INFORMATION FROM THE CALLED TASK TO THE CALLING TASK.

BULLET 6: THE FIRST ENTRY CALL IS IN POSITIONAL NOTATION AND THE SECOND IS IN NAMED NOTATION. THIS IS AN ARBITRARY CHOICE FOR PURPOSES OF ILLUSTRATION. THE FIRST ENTRY CALL CALLS THE Receive_Message ENTRY OF TASK OBJECT Message_Buffer_1 TO PLACE A VALUE PRODUCED BY THAT TASK OBJECT IN M. (THE PARAMETER MODE, AS SHOWN IN THE CONTEXT, IS <u>out</u>.) THE SECOND ENTRY CALL CALLS THE Send_Message ENTRY OF TASK OBJECT Message_Buffer_2 TO PASS THE VALUE OF M TO THAT TASK OBJECT. (THE PARAMETER MODE IS <u>in</u>.)

AGAIN, ALERT THOSE HAVING TROUBLE WITH TASK OBJECTS AS DATA TO THIS EXAMPLE.

ENTRY CALLS

- ENTRIES OF A TASK CAN BE CALLED AS IF THEY WERE SUBPROGRAMS

- USUAL FORM:

  ```
  task object name . entry name [( actual parameters )];
  ```

- ACTUAL PARAMETERS MAY BE NAMED OR POSITIONAL.

- MAY BE USED TO PASS INFORMATION TO OR FROM THE EXECUTING TASK OBJECT WHOSE ENTRY IS CALLED

  -- THE TASK <u>ACCEPTS</u> THE ENTRY CALL BY EXECUTING AN ACCEPT STATEMENT

  -- PARAMETER MODES HAVE THEIR USUAL MEANINGS

- CONTEXT FOR EXAMPLE:

```
task type Message_Buffer_Type is
   entry Send_Message (Message : in Message_Type);
   entry Receive_Message (Message : out Message_Type);
end Message_Buffer_Type;

Message_Buffer_1, Message_Buffer_2 : Message_Buffer_Type;
M : Message_Type;
```

- EXAMPLE:

```
Message_Buffer_1.Receive_Message (M);
Message_Buffer_2.Send_Message (Message =>  M);
```

INSTRUCTOR NOTES

THE "TYPICAL FORM" DESCRIBED IN BULLET 2 DOES NOT ACCOUNT FOR ENTRY FAMILIES OR ACCEPT STATEMENTS NOT CONTAINING SEQUENCES OF STATEMENTS. NEITHER IS IT INDICATED THAT THE NAME OF THE END IS OPTIONAL.

THOUGH, AS BULLET 4 INDICATES, AN ACCEPT DOES NOT NORMALLY HAVE A DECLARATIVE PART, THE SQUARE OF STATEMENTS MAY CONSIST OF A SINGLE BLOCK STATEMENT WITH ITS OWN DECLARATIVE PART.

ACCEPT STATEMENTS

- OCCUR ONLY IN TASK BODIES, AND SPECIFY WHAT TO DO WHEN AN ENTRY OF THE TASK IS CALLED.

- TYPICAL FORM:

```
accept [entry name] [( [formal parameters] )] do
   [sequence of statements]
end [entry name] ;
```

- SIMILAR IN SOME WAYS TO A PROCEDURE BODY

  -- FORMAL PARAMETERS HAVE THE SAME STRUCTURE, AND MAY BE REFERRED TO ONLY INSIDE THE ACCEPT STATEMENT

  -- AN ACCEPT STATEMENT MAY BE LEFT BY MEANS OF A RETURN STATEMENT

- IMPORTANT DIFFERENCES:

  -- THE ACCEPT STATEMENT IS EXECUTED WHEN THE CALLED TASK GETS TO IT
  -- THERE MAY BE SEVERAL DISTINCT ACCEPT STATEMENTS FOR THE SAME ENTRY
  -- AN ACCEPT STATEMENT HAS NO DECLARATIVE PART

INSTRUCTOR NOTES

Word_Task AND Output_Line_Task ARE TASKS 2 AND 3, RESPECTIVELY, OF THE TEXT REFORMATTER DEPICTED NEAR THE BEGINNING OF THE SECTION ON TASKS. (NOTE THAT THESE ARE NOT TASK TYPES, BUT TASK OBJECTS BELONGING TO ANONYMOUS TASK TYPES.)

Word_Task ACCEPTS A SEQUENCE OF BLANK AND NON-BLANK CHARACTERS, GROUPS CONSECUTIVE NON-BLANK CHARACTERS INTO WORDS, AND CALLS THE ENTRY Output_Line_Task.Deliver_Word WITH EACH WORD FOUND.

THERE ARE TWO ACCEPT STATEMENTS FOR THE Deliver_Character ENTRY. THE ONE IN THE FIRST INNER LOOP HANDLES CALLS ON Deliver_Character MODE WHILE Word_Task IS SCANNING A SEQUENCE OF ONE OR MORE BLANKS. THE ONE IN THE SECOND INNER LOOP HANDLES CALLS ON Deliver_Character MODE WHILE Word_Task IS SCANNING A SEQUENCE OF ONE OR MORE NON-BLANKS. THE TASK CALLING Word_Task.Deliver_Character IS OBLIVIOUS TO WHERE CONTROL IS WITHIN Word_Task AND WHICH ACCEPT STATEMENT WILL HANDLE THE CALL.

EXAMPLE OF ACCEPT STATEMENTS

CONTEXT:

```
task Output_Line_Task is
   entry Deliver_Word (Word : in String);
end Output_Line_Task;
```

EXAMPLE:

```
task Word_Task is
   entry Deliver_Character (Char : in Character);
end Word_Task;
task body Word_Task is
   Next_Character : Character;
   Word           : String (1 .. 65);
   Next_Position  : Integer range 1 .. 66;
begin -- Word_Task
   Word_Loop:
      loop
         Blank_Loop:
            loop
               accept Deliver_Character (Char : in Character) do
                  Next_Character := Char;
               end Deliver_Character;
               exit Blank_Loop when Next_Character /= ' ';
            end loop Blank_Loop;
         Next_Position := 1;
         Non_Blank_Loop:
            loop
               Word (Next_Position) := Next_Character;
               Next_Position := Next_Position + 1;
               accept Deliver_Character (Char : in Character) do
                  Next_Character := Char;
               end Deliver_Character;
               exit Non_Blank_Loop when Character = ' ';
            end loop Non_Blank_Loop;
         Output_Line_Task.Deliver_Word (Word (1 .. Next_Position - 1));
      end loop Word_Loop;
end Word_Task;
```

INSTRUCTOR NOTES

IN A RENDEZVOUS BETWEEN TWO PEOPLE, WHOEVER ARRIVES FIRST AT THE AGREED-UPON MEETING PLACE WAITS FOR THE OTHER.

THE PLURAL OF <u>RENDEZVOUS</u> IS <u>RENDEZVOUS</u>. THE SINGULAR OF <u>RENDEZVOUS</u> IS <u>RENDEZVOUS</u>.

RENDEZVOUS

- AN ENTRY CALL IS EXECUTED WHEN THE CALLING TASK AND CALLED TASK HAVE BOTH ARRIVED AT A POINT WHERE THEY EXPECT COMMUNICATION TO TAKE PLACE. THIS IS CALLED A RENDEZVOUS.

  -- IF THE CALLED TASK REACHES AN ACCEPT STATEMENT FIRST, IT WAITS FOR SOME TASK TO CALL THE ENTRY.

  -- IF A TASK ISSUES AN ENTRY CALL BEFORE THE CALLED TASK REACHES AN ACCEPT STATEMENT, THE CALLING TASK WAITS.

  -- SEVERAL DIFFERENT TASKS MIGHT CALL THE SAME ENTRY OF THE SAME TASK BEFORE THE CALLED TASK REACHES AN ACCEPT STATEMENT FOR THAT ENTRY. EACH CALLING TASK WAITS, AND ENTRY CALLS ARE ACCEPTED IN ORDER OF ARRIVAL.

- STEPS IN A RENDEZVOUS:

  -- THE in AND in out ACTUAL PARAMETERS OF THE ENTRY CALL ARE COPIED INTO THE FORMAL PARAMETERS OF THE ACCEPT STATEMENT.

  -- THE ACCEPT STATEMENT IS EXECUTED.

  -- THE in out AND out FORMAL PARAMETERS OF THE ACCEPT STATEMENT ARE COPIED BACK TO THE ACTUAL PARAMETERS OF THE ENTRY CALL, COMPLETING EXECUTION OF THE ENTRY CALL STATEMENT.

- FOLLOWING A RENDEZVOUS, BOTH TASKS RESUME ASYNCHRONOUS EXECUTION.

INSTRUCTOR NOTES

BULLET 1: MORE GENERAL FORMS OF THE SELECT STATEMENT ARE GIVEN LATER. THE ACCEPT STATEMENTS NEED NOT BE FOR DIFFERENT ENTRIES (SEE BULLET 3).

BULLET 2: IT IS PREFERABLE TO DO AS MUCH OF THE PROCESSING AS POSSIBLE IN THE SEQUENCE OF STATEMENTS TO FOLLOW THE ACCEPT STATEMENT RATHER THAN IN THE ACCEPT STATEMENT, SINCE THE CALLING TASK CANNOT PROCEED ON ITS OWN UNTIL THE ACCEPT STATEMENT IS COMPLETE.

BULLET 3: Ada DOES NOT SPECIFY HOW AN ACCEPT STATEMENT IS CHOSEN, BUT THE METHOD SHOULD GIVE EACH ONE A FAIR CHANCE OF BEING CHOSEN. THE ORDER IN WHICH ACCEPT STATEMENTS ARE LISTED AND THE ORDER IN WHICH CALLS ON DIFFERENT ENTRIES ARRIVE ARE BOTH IRRELEVANT. (ON THE OTHER HAND, ONCE AN ENTRY HAS BEEN SELECTED, IT IS ALWAYS THE LARGEST WAITING CALL ON THAT ENTRY THAT IS ACCEPTED FIRST.)

SELECTIVE WAITS

- A STATEMENT OF THE FORM

```
select
   accept statement
   sequence of zero or more statements
{ or
   accept statement
   sequence of zero or more statements }
end select;
```

  TELLS THE CALLING TASK TO WAIT FOR A CALL ON ANY OF THE ENTRIES AND ACCEPT WHICHEVER ONE OCCURS FIRST.

- AFTER THE RENDEZVOUS, THE SEQUENCE OF STATEMENTS FOLLOWING THE CHOSEN ACCEPT STATEMENT IS EXECUTED.

- IF CALLS ARE ALREADY WAITING FOR TWO OR MORE ACCEPT STATEMENTS, WHEN THE SELECT STATEMENT IS ENCOUNTERED, ONE OF THESE ACCEPT STATEMENTS IS CHOSEN <u>ARBITRARILY</u>.

INSTRUCTOR NOTES

OBJECTS IN THIS TASK TYPE LOOP FOREVER, ACCEPTING INTERLEAVED CALLS ON THE TWO ENTRIES. CALLS ON Increase_Count ARE HANDLED (AMONG THEMSELVES) IN ORDER OF ARRIVAL, AS ARE CALLS ON Get_Count. HOWEVER, THE SEQUENCE OF ACCEPTANCES OF Increase_Count CALLS AND THE SEQUENCE OF ACCEPTANCES OF Get_Count CALLS MAY BE INTERLEAVED IN ANY MANNER.

<u>TASK OBJECTS AS DATA</u>: AN OBJECT OF TYPE Shared_Count_Type HAS A NON-NEGATIVE COUNT ASSOCIATED WITH IT, INITIALLY ZERO. THE OPERATION Shared_Count_Object.Increase_Count (n) INCREASES THE COUNT ASSOCIATED WITH Shared_Count_Object BY n. THE OPERATION Shared_Count_Object.Get_Count (x) SETS x TO THE COUNT ASSOCIATED WITH Shared_Count_Object.

THESE OPERATIONS CAN BE INVOKED BY SEVERAL OTHER TASKS EXECUTING ASYNCHRONOUSLY. BECAUSE A GIVEN Shared_Count_Type TASK OBJECT CAN BE EXECUTING AT MOST ONE ACCEPT STATEMENT AT A GIVEN TIME, AT MOST ONE OPERATION ON THE Shared_Count_Type OBJECT CAN BE IN PROGRESS AT ONE TIME. THUS A Shared_Count_Type OBJECT IS ONE THAT CAN BE OPERATED ON BY ASYNCHRONOUS TASKS WITHOUT FEAR THAT OPERATIONS INVOKED BY SEVERAL TASKS WILL INTERFERE WITH EACH OTHER.

IN SOME TASKING APPLICATIONS, INFINITE LOOPS MAKE PERFECT SENSE. HOWEVER, THAT IS PROBABLY NOT THE CASE HERE. THE NEXT SLIDE PRESENTS A SOLUTION TO THIS PROBLEM.

EXAMPLE OF A SELECTIVE WAIT

```
task type Shared_Count_Type is
   entry Increase_Count (Increment : in Positive);
   entry Get_Count (Sum_So_Far : out Natural);
end Shared_Count_Type;

task body Shared_Count_Type is

   Sum : Natural := 0;

begin -- Shared_Count_Type

   loop

      select

         accept Increase_Count (Increment : in Positive) do
            Sum := Sum + Increment;
         end Increase_Count;

      or

         accept Get_Count (Sum_So_Far : out Natural) do
            Sum_So_Far := Sum;
         end Get_Count;

      end select;

   end loop;

end Shared_Count_Type;
```

INSTRUCTOR NOTES

BULLET 2: RULES FOR DEPARTURE FROM A UNIT AWAITING TERMINATION OF SOME TASK WHERE GIVEN AND ILLUSTRATED ON TWO EARLIER SLIDES.

BULLET 3: ONCE ALL THESE TERMINATE ALTERNATIVES ARE CHOSEN SIMULTANEOUSLY, THE SUBPROGRAM BODY, BLOCK STATEMENT, OR TASK BODY REFERRED TO IN BULLET 2 CAN BE DEPARTED FROM.

BULLET 4: WHEN THE DESCRIBED CONDITIONS OCCUR, THERE IS NO TASK LEFT IN A POSITION TO CALL THE ENTRIES OF THE ABOUT-TO-BE-TERMINATED TASK.

BULLET 5: THIS LOOP IS TAKEN FROM THE TASK BODY ON THE PREVIOUS SLIDE AND AUGMENTED WITH A TERMINATE ALTERNATIVE.

THE terminate ALTERNATIVE

- A SELECTIVE WAIT MAY HAVE ONE ALTERNATIVE CONSISTING ONLY OF THE RESERVED WORD terminate. IT IS USED TO SPECIFY SIMULTANEOUS TERMINATION OF A GROUP OF RELATED TASKS.

- THE TERMINATE ALTERNATIVE IS CHOSEN BY TASK OBJECT T UNDER THE FOLLOWING CIRCUMSTANCES:

  -- SOME SUBPROGRAM BODY, BLOCK STATEMENT, OR TASK BODY HAS FINISHED ITS OWN WORK, BUT DEPARTURE FROM ITS SEQUENCE OF STATEMENTS IS AWAITING TERMINATION OF TASK OBJECT T AND POSSIBLY SOME OTHER TASKS

  -- ALL THESE OTHER TASKS ARE ALSO WAITING AT A SELECTIVE WAIT WITH A TERMINATE ALTERNATIVE.

- THEN THE TERMINATE ALTERNATIVE IS CHOSEN BY EACH OF THE RELATED TASKS, CAUSING THE TASKS TO TERMINATE.

- INTENT IS TO ALLOW A TASK TO TERMINATE WHEN ALL ITS WORK IS DONE.

- EXAMPLE OF A SELECTIVE WAIT WITH A TERMINATE ALTERNATIVE:

```
loop
   select
      accept Increase_Count (Increment : in Positive) do
         Sum := Sum + Increment;
      end
   or
      accept Get_Count (Sum_So_Far : out Natural) do
         Sum_So_Far := Sum;
      end Get_Count;
   or
      terminate;
   end select;
end loop;
```

INSTRUCTOR NOTES

BULLET 1: A TERMINATE ALTERNATIVE MAY HAVE A GUARD.

BULLET 2: ASSUME Queue_Package_Template IS A GENERIC PACKAGE PROVIDING A TYPE Queue_Type FOR A QUEUE IMPLEMENTED AS A CIRCULAR LIST. THE TYPE OF THE QUEUE ELEMENTS AND THE CAPACITY OF THE QUEUE ARE GENERIC PARAMETERS. OPERATIONS INCLUDE Is_Full, Is_Empty, Enqueue, AND Dequeue, WITH THE OBVIOUS MEANINGS.

CALLS ON Send_Message CAN ONLY BE ACCEPTED DURING ITERATIONS OF THE LOOP FOR WHICH THE QUEUE IS NOT FULL. CALLS ON Receive_Message CAN ONLY BE ACCEPTED DURING ITERATIONS OF THE LOOP FOR WHICH THE QUEUE IS NOT EMPTY.

A Message_Buffer_Type OBJECT HAS TWO OPERATIONS, SENDING A MESSAGE TO THE BUFFER AND RECEIVING A MESSAGE FROM THE BUFFER. ONCE A TASK HAS SENT A MESSAGE TO THE BUFFER, IT CAN NORMALLY PROCEED WITHOUT WAITING FOR THE MESSAGE TO BE RECEIVED. HOWEVER, IF THE BUFFER IS FULL, THE FIRST GUARD PREVENTS THE ENTRY CALL ON Send_Message FROM BEING ACCEPTED, SO THE CALLING TASK WAITS UNTIL THERE IS ROOM IN THE BUFFER. A TASK RECEIVING A MESSAGE FROM THE BUFFER NEED NOT WAIT IF THERE IS ALREADY A MESSAGE WAITING FOR IT. IF NO MESSAGE IS WAITING, HOWEVER, THE SECOND GUARD PREVENTS ACCEPTANCE OF THE CALL ON Receive_Message, SO THE CALLING TASK WAITS UNTIL A MESSAGE ARRIVES.

GUARDS ON SELECTIVE WAITS

- AN ALTERNATIVE IN A SELECTIVE WAIT MAY BE PRECEDED BY A BOOLEAN EXPRESSION CALLED A <u>GUARD</u>. ALL GUARDS ARE EVALUATED WHEN A SELECTIVE WAIT IS ENCOUNTERED, AND ONLY THE ALTERNATIVES WITH TRUE GUARDS ARE ELIGIBLE FOR SELECTION.

- EXAMPLE:

```
task type Message_Buffer_Type is
   entry Send_Message (Message : in Message_Type);
   entry Receive_Message (Message : out Message_Type);
end Message_Buffer_Type;

task body Message_Buffer_Type is
   package Message_Queue_Package is new
      Queue_Package_Template (Capacity =>  10, Item_Type =>  Message_Type);
   Queue : Message_Queue_Package.Queue_Type;
begin -- Message_Buffer_Type
   loop
      select
         when not Message_Queue_Package.Is_Full (Queue) =>
            accept Send_Message (Message : in Message_Type) do
               Message_Queue_Package.Enqueue (Message, Queue);
            end Send_Message;
      or
         when not Message_Queue_Package.Is_Empty (Queue) =>
            accept Receive_Message (Message : out Message_Type) do
               Message_Queue_Package.Dequeue (Message, Queue);
            end Receive_Message;
      or
         terminate;
      end select;
   end loop;
end Message_Buffer_Type;
```

INSTRUCTOR NOTES

FROM THIS POINT ON, THE PACE OF THE PRESENTATION SHOULD QUICKEN. REMAINING TOPICS ARE COVERED IN LESS DETAIL.

BULLET 1: A DELAY STATEMENT NORMALLY CAUSES EXECUTION OF A TASK TO BE DELAYED FOR AT LEAST THE SPECIFIED NUMBER OF SECONDS. AT THE BEGINNING OF AN ALTERNATIVE IN A SELECT STATEMENT, IT SPECIFIES THAT IF NO OTHER ALTERNATIVE HAS BEEN CHOSEN WITHIN THE GIVEN NUMBER OF SECONDS, THAT ALTERNATIVE SHOULD BE CONSIDERED TO HAVE BEEN SELECTED, AND THE DELAY STATEMENT SHOULD BE CONSIDERED TO HAVE JUST BEEN EXECUTED. THIS SELECT STATEMENT UPDATES Current_Velocity AND Current_Position WITH FRESH DATA IF THAT DATA ARRIVES WITHIN 0.5 SECONDS, OR PROJECTS A NEW Current_Position BASED ON Current_Velocity OTHERWISE.

A SELECT STATEMENT MAY HAVE SEVERAL ACCEPT ALTERNATIVES AND SEVERAL DELAY ALTERNATIVES. ONLY THE SHORTEST DELAY HAS ANY EFFECT.

BULLET 2: A SELECT STATEMENT MAY HAVE AN ELSE-PART, CHOSEN IF NO OTHER ALTERNATIVE IS READY TO BE EXECUTED WHEN THE SELECT STATEMENT IS ENCOUNTERED. THIS EXAMPLE ALWAYS EXECUTES Normal_Processing_Routine, EXCEPT WHEN THERE IS A PENDING CALL ON ONE OF THE ENTRIES Report_Critical_Situation_1 AND Report_Critical_Situation_2.

DELAY ALTERNATIVES, ELSE-PARTS, AND TERMINATE ALTERNATIVES ARE MUTUALLY EXCLUSIVE.

BULLET 3: NORMALLY SELECT ALTERNATIVES BEGIN WITH ACCEPT STATEMENTS RATHER THAN ENTRY CALLS. TIMED AND CONDITIONAL ENTRY CALLS ARE EXCEPTIONS TO THIS RULE. THERE IS NO FORM OF THE SELECT STATEMENT ALLOWING ONE OF SEVERAL ENTRY CALL STATEMENTS TO BE SELECTED ARBITRARILY.

OTHER FORMS OF SELECT STATEMENTS

- A SPECIAL ALTERNATIVE CHOSEN IF NO ENTRY CALL ARRIVES IN A SPECIFIED AMOUNT OF TIME:

```
select
   accept Report_Navigation_Data
      (Velocity : in Velocity_Type; Position : in Position_Type) do
       Current_Velocity := Velocity;
       Current_Position := Position;
   end Report_Navigation_Data;
or
   delay 0.5;
   Current_Position := Projected_Position (Current_Position, Current_Velocity);
end select;
```

- A SPECIAL ALTERNATIVE CHOSEN IF NO ENTRY CALL IS <u>ALREADY</u> WAITING:

```
select
   accept Report_Critical_Situation_1 (Location : in Location_Type) do
      Reported_Location := Location;
   end Report_Critical_Situation_1;
   Critical_Processing_Routine_1 (Reported_Location);
or
   accept Report_Critical_Situation_2 (Location : in Location_Type) do
      Reported_Location := Location;
   end Report_Critical_Situation_2;
   Critical_Processing_Routine_2 (Reported_Location);
else
   Normal_Processing_Routine;
end select;
```

- SIMILAR FORMS FOR <u>TIMED ENTRY CALLS</u> AND <u>CONDITIONAL ENTRY CALLS</u>

  -- A SINGLE <u>ENTRY CALL</u> ENCLOSED IN A SELECT STATEMENT WITH A DELAY ALTERNATIVE OR ELSE-PART.

INSTRUCTOR NOTES

BULLET 1: COVER THIS QUICKLY, WITHOUT GOING INTO DETAIL.

BULLET 2: THIS ATTRIBUTE MUST BE USED WITH CAUTION. ITS LOGICAL VALUE CAN INCREASE AS NEW ENTRY CALLS ARRIVE OR DECREASE AS TIMED ENTRY CALLS EXPIRE AFTER THE ATTRIBUTE HAS BEEN EVALUATED.

BULLET 3: AN IMPLEMENTATION DEFINES WHAT STORAGE IS CONSIDERED USED BY A "TASK OBJECT" AND WHAT STORAGE IS CONSIDERED USED BY ITS "ACTIVATION." THE INTENT SEEMS TO BE THAT THE STORAGE ALLOCATED AS A TASK'S DATA IS USED BY ITS "ACTIVATION" WHILE TASK CONTROL INFORMATION IS USED BY THE "OBJECT."

BULLET 4: THE ENTRY E IS KNOWN AS AN <u>INTERRUPT ENTRY</u>. THIS PROVIDES A HIGH-LEVEL VIEW OF HARDWARE INTERRUPTS, AND IS USED IN WRITING INTERRUPT HANDLING TASKS (SUCH AS I/O DEVICE HANDLERS).

ATTRIBUTES FOR USE WITH TASKS

- BOOLEAN-VALUED ATTRIBUTES TO DETERMINE THE PROGRESS OF A TASK OBJECT T:

T'Callable :	False IF THE TASK OBJECT IS DONE EXECUTING (BUT POSSIBLY AWAITING THE CONCLUSION OF OTHER TASKS); True OTHERWISE.
T'Terminated:	True IF T IS DONE EXECUTING AND NO LONGER AWAITING THE CONCLUSION OF ANY OTHER TASK; False OTHERWISE.

- THE NUMBER OF CALLS ON ENTRY T.E WAITING TO BE ACCEPTED:

  T.E'Count

- REPRESENTATION ATTRIBUTES FOR A TASK OBJECT T OR TASK TYPE T:

T'Storage_Size :	NUMBER OF STORAGE UNITS RESERVED FOR TASK <u>ACTIVATIONS</u> (INCLUDING DATA AREAS)
T'Size :	NUMBER OF BITS IN A TASK <u>OBJECT</u>
T'Address :	ADDRESS OF FIRST MACHINE INSTRUCTION FOR THE TASK

- ADDRESS CLAUSE TO ASSOCIATE AN ENTRY E WITH A HARDWARE INTERRUPT:

  for E'Address use | expression of type System.Address | ;

  -- GOES IN A TASK DECLARATION

  -- MAKES THE OCCURRENCE OF A SPECIFIED INTERRUPT LOOK LIKE A CALL ON ENTRY E

  -- ASSOCIATION BETWEEN INTERRUPTS AND ADDRESSES IS IMPLEMENTATION-DEFINED

INSTRUCTOR NOTES

IN THIS EXAMPLE, THE DIFFERENT ENTRIES IN THE FAMILY ARE USED FOR ENTRY CALLS WITH DIFFERENT PRIORITIES DEFINED BY THE PROGRAMMER. CALLS ON EACH INDIVIDUAL ENTRY ARE SERVICED ON A FIRST-COME FIRST-SERVED BASIS, BUT THE ACCEPTING LIST CAN CONTROL THE ORDER IN WHICH IT ACCEPTS CALLS ON DIFFERENT ENTRIES IN THE FAMILY.

(THE 'Count ATTRIBUTES IN THE GUARDS HAVE ENTRY NAMES NOT QUALIFIED BY TASK OBJECT NAMES. THIS IS ALLOWED WITHIN A TASK BODY AND DENOTES AN ENTRY OF THE TASK OBJECT EXECUTING THE STATEMENT.)

(THE DELAY ALTERNATIVE IS PROVIDED SO THAT IF, FOR EXAMPLE, A PENDING TIMED ENTRY CALL ON Schedule_Job (1) CAUSES BOTH GUARDS TO EVALUATE TO False, BUT THE TIMED ENTRY CALL THEN EXPIRES BEFORE THE THIRD ALTERNATIVE (THE ONLY REMAINING ELIGIBLE ALTERNATIVE) CAN BE SELECTED, CALLS ON THE OTHER TWO ENTRIES ARE NOT BLOCKED FOR MORE THAN ONE SECOND).

ENTRY FAMILIES

- AN <u>ENTRY FAMILY</u> IS A ONE-DIMENSIONAL "ARRAY" OF ENTRIES.
- A TASK DECLARATION MAY CONTAIN AN ENTRY DECLARATION LIKE THE FOLLOWING:

```
entry Schedule_Job (1 .. 3) (Job : in Job_Description_Type);
```

- IN EFFECT, THIS DECLARES THREE ENTRIES:

```
Schedule_Job (1), Schedule_Job (2), Schedule_Job (3)
```

- CALLS ON A MEMBER OF THE ENTRY FAMILY SPECIFY AN INDEX:

$$\underbrace{\underbrace{\texttt{Job\_Scheduler.Schedule\_Job}}_{\text{ENTRY FAMILY NAME}}\ \underbrace{\texttt{(Priority\_Level)}}_{\text{INDEX}}}_{\text{NAME OF A SINGLE ENTRY}}\ \underbrace{\texttt{(Job Description)}}_{\text{ACTUAL PARAMETER}}$$

- ACCEPT STATEMENTS FOR MEMBERS OF AN ENTRY FAMILY ALSO SPECIFY AN INDEX:

```
loop
   select
      when Schedule_Job (1)'Count = 0 and Schedule_Job (2)'Count = 0 =>
         accept Schedule_Job (3) (Job : in Job_Description_Type) do
            ...
         end Schedule_Job;
   or
      when Schedule_Job (1)'Count = 0 =>
         accept Schedule_Job (2) (Job : in Job_Description_Type) do
            ...
         end Schedule_Job;
   or
      accept Schedule_Job (1) (Job : in Job_Description_Type) do
         ...
      end Schedule_Job;
   or
      delay 1.0;
   end select;
end loop;
```

INSTRUCTOR NOTES

A "PLEASE TERMINATE" ENTRY HAS TO BE DESIGNED INTO A TASK FROM THE BEGINNING. UPON ACCEPTING A CALL ON THIS ENTRY, A TASK PERFORMS APPROPRIATE CLEANUP OPERATIONS, SUCH AS CLOSING FILES, TURNING OFF DEVICES, OR DEALLOCATING VARIABLES, THEN COMPLETES ITS SEQUENCE OF STATEMENTS.

PRIORITIES ARE SPECIFIED ON A TYPE-BY-TYPE BASIS.

THE EFFECT OF A PRIORITY PRAGMA DEPENDS ON THE NUMBER OF PROCESSORS AVAILABLE TO EXECUTE TASKS.

OTHER CONTROLS ON TASK EXECUTION

- THE <u>abort</u> STATEMENT: abort $\boxed{\text{task name}}$;
  - -- IMMEDIATELY CAUSES THE NAMED TASK TO BECOME COMPLETED.
  - -- IT IS BETTER TO CALL AN ENTRY DESIGNED TO RECEIVE REQUESTS TO TERMINATE. THIS GIVES THE TASK A CHANCE TO RELEASE RESOURCES AND FINISH IN A WELL-DEFINED STATE.
  - -- THE abort STATEMENT SHOULD BE RESERVED FOR TASKS THAT HAVE GONE OUT OF CONTROL AND DO NOT RESPOND TO POLITE REQUESTS TO TERMINATE.

- THE PRIORITY PRAGMA: pragma Priority ($\boxed{\text{Priority\_Level}}$);
  - -- THE PRIORITY LEVEL IS AN EXPRESSION BELONGING TO System.Priority, A *PREDEFINED SUBTYPE OF TYPE Integer*.
  - -- INDICATE DEGREES OF URGENCY FOR TASKS (LOWER NUMBER = LESS URGENCY)
  - -- USED TO DETERMINE WHICH TASKS WILL BE PROCESSED WHEN THE NUMBER OF TASKS THAT COULD SENSIBLY BE EXECUTING AT A GIVEN MOMENT IS GREATER THAN THE NUMBER OF PROCESSORS.
  - -- NOT TO BE USED FOR SCHEDULING:
    - A TASK WITH HIGHER PRIORITY DOES NOT NECESSARILY BLOCK A TASK WITH LOWER PRIORITY FROM EXECUTING.
    - A TASK WITH HIGHER PRIORITY CAN BE WAITING (FOR AN ENTRY CALL TO BE ISSUED OR ACCEPTED, OR FOR A DELAY STATEMENT OR I/O OPERATION TO COMPLETE) WHILE A TASK WITH LOWER PRIORITY EXECUTES.

INSTRUCTOR NOTES

(THE OPTIMIZATION CONSISTS OF KEEPING A "COPY" OF THE SHARED VARIABLE IN A TASK'S OWN REGISTER, AND READING AND UPDATING THAT COPY BETWEEN RENDEZVOUS. THE VALUE IS COPIED BACK JUST BEFORE A RENDEZVOUS OR TASK TERMINATION.)

THE PRAGMA GOES IN THE SAME SEQUENCE OF DECLARATIONS AS THE DECLARATION OF THE SHARED VARIABLE.

SHARED VARIABLES

- SEVERAL TASKS CAN REFER TO THE SAME GLOBAL VARIABLE, BUT IT IS DANGEROUS. INTERLEAVED MANIPULATIONS OF THE SAME VARIABLE CAN INTERFERE WITH EACH OTHER.

- NORMALLY, DATA USED BY TWO OR MORE TASKS SHOULD BE DECLARED INSIDE A NEW TASK, CALLED A MONITOR TASK.

  -- OPERATIONS ON THE DATA CAN ONLY BE PERFORMED BY CALLING ENTRIES OF THE MONITOR TASK.

  -- THIS GUARANTEES THAT ONLY ONE OPERATION CAN BE IN PROGRESS AT A GIVEN TIME (SINCE THE MONITOR TASK WILL BE ACCEPTING AT MOST ONE ENTRY AT A GIVEN TIME).

- A COMPILER HAS THE RIGHT TO PERFORM CERTAIN OPTIMIZATIONS FOR A SCALAR OR ACCESS-TYPE GLOBAL VARIABLE USED IN A TASK, BASED ON THE ASSUMPTION THAT NO OTHER TASK IS USING THE SAME GLOBAL VARIABLE.

  -- IF THIS ASSUMPTION IS NOT TRUE, THERE MAY BE UNEXPECTED RESULTS.

  -- THE OPTIMIZATION CAN BE PREVENTED BY THE FOLLOWING PRAGMA:

```
pragma Shared ( shared variable name );
```

Material: Advanced Ada Topics (L305), Volume III

We would appreciate your comments on this material and would like you to complete this brief questionaire. The completed questionaire should be forwarded to the address on the back of this page. Thank you in advance for your time and effort.

1. Your name, company or affiliation, address and phone number.

2. Was the material accurate and technically correct?

   Yes ☐ No ☐

   Comments:

3. Were there any typographical errors?

   Yes ☐ No ☐

   If yes, on what pages?

4. Was the material organized and presented appropriately for your applications?

   Yes ☐ No ☐

   Comments:

5. General Comments: